VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

XQuery algebra
XQuery Algebra

2012

Bc. Petr Lukáš

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Petr Lukáš**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: XQuery algebra
XQuery Algebra

Zásady pro vypracování:

Cílem této práce je implementace XQuery algebry založené na stromové reprezentaci mezivýsledku.

Práce bude mít následující atributy:
1. Vstupní normalizovaný XQuery dotaz je převeden do stromové reprezentace.
2. Implementace pravidel, které umožní přepis jedné reprezentace do další ekvivalentní.
3. Implementace jednoduchých fyzických operátorů jako je třídění, selekce, projekce, kartézský součin a běžný join.
4. Implementace XQuery procesoru, který bude volat jednotlivé fyzické operátory.

Seznam doporučené odborné literatury:

Christopher Re, Jerome Simeon, Mary Fernandez. A Complete and Efficient Algebraic Compiler for XQuery, 22nd International Conference on Data Engineering (ICDE'06), 2006, IEEE DL

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Radim Bača, Ph.D.**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literální prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2012

Chtěl bych poděkovat svému vedoucímu diplomové práce, panu Ing. Radimu Bačovi, Ph.D za poskytnutí velmi zajímavého tématu a za průběžnou ochotnou odbornou pomoc.

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací procesoru dotazovacího jazyka XQuery, který slouží k prohledávání stromově organizovaných XML dokumentů a databází. Cílem je navrhnout a vytvořit procesor pracující na principu algebraických operátorů, které umožňují efektivnější vyhodnocování vstupních dotazů než přímá interpretace.

V úvodu práce je podán stručný přehled aktuálně používaných technologií, které se k dotazování nad XML úzce vážou, včetně krátkého popisu samotného jazyka XQuery. Následuje návrh algebraických operátorů a návrh překladových pravidel pro transformaci dotazů na tyto operátory zahrnující také optimalizační postupy, které umožňují v přeloženém dotazu provést určité úpravy tak, aby výsledek zůstal zachován, ale výpočet proběhl efektivněji. Teoretické návrhy jsou dále převedeny do skutečné podoby ve formě implementace procesoru. V závěru práce je výsledný procesor porovnán s jinými existujícími a běžně používanými implementacemi.

Klíčová slova: XML, XQuery, XPath, dotazování nad XML, dotazovací jazyky, procesor, algebra, optimalizace, operátor, plán dotazu, překladač

Abstract

This diploma thesis deals with design and implementation of a processor of the XQuery computer language used for searching data in tree organized XML documents and databases. The goal of this work is to design and create a processor based on algebraic operators which can give better performance in evaluation of the input queries than direct interpretation.

At the beginning of this work a breaf overview of currently used technologies for querying XML including a short description of the XQuery language is given. It is followed by a proposal of algebraic operators and proposal of compilation rules transforming queries into those operators. There are also included optimization techniques enabling to make such modifications in the compiled query as the result is preserved, but the evaluation is more effective. Those proposals are subsequently transformed into the real form of processor implementation. The final part of this work compares the implemented processor to other existing and commonly used implementations.

Keywords: XML, XQuery, XPath, querying XML, data query languages, processor, algebra, optimization, operator, query plan, compiler

Seznam použitých zkratek a symbolů

DOM	–	Document Object Model
DQL	–	Data Query Languages
EBNF	–	Extended Backus–Naur Form
HTML	–	HyperText Markup Language
PDF	–	Portable File Format
PI	–	Processing Instruction
SOAP	–	Simple Object Access Protocol
SQL	–	Structured Query Language
TPNF	–	Tree Pattern Normal Form
TPQ	–	Tree Pattern Query
XML	–	eXtensible Markup Language
XSLT	–	eXtensible Stylesheet Language Transformation

Obsah

Seznam obrázků

Seznam tabulek

Seznam algoritmů

1 Úvod

Jedním z nejvíce prakticky uplatňovaných oborů dnešních informačních technologií je tvorba informačních systémů. Zejména v současné době nutí politická situace přecházet menší či větší firmy na komplexní informační systémy obvykle z důvodu zefektivnění výroby nebo služeb.

Je důležité umět shromáždit potřebná data, najít mezi nimi patřičné vztahy a co je hlavní, je nutné v nich umět rychle vyhledávat. Dotazovací jazyky (DQL - Data Query Languages) jsou vyvíjeny již celou řadu let a zaměřují se především na relační databáze, kde jsou data organizovány v tzv. relacích nebo zjednodušeně řečeno v tabulkách. Jednoznačně nejpoužívanějším představitelem těchto jazyků je SQL (Structured Query Language).

Relace ale nejsou jedinou možností jak organizovat data. Trendem moderní doby je stále více se uplatňující stromová struktura v podání XML, která je v mnoha případech schopna lépe a pružněji reprezentovat určitou reálnou situaci. S tím roste potřeba umět v takovýchto stromových organizacích efektivně a hlavně snadno vyhledávat.

Zatímco v relačních databázích má jazyk SQL své místo více neohrozitelné a vývojáři informačních systémů se bez něj neobejdou, v případě XML databází jsou vzniklé standardy poměrně čerstvé a do podvědomí veřejnosti teprve postupně pronikají. Už v tuto chvíli lze ale říci, že největšími kandidáty na široké uplatnění jsou XPath a XQuery. Je vhodné předeslat, že XQuery je v uvozovkách pouze rozšířením jazyka XPath, čili že se nejedná o dva naprosto odlišné jazyky.

Cílem práce je naimplementovat zjednodušený procesor poměrně náročného deklarativního jazyka XQuery za použití algebraických operátorů. Práce se bude skládat ze čtyř hlavních kapitol. První z nich, Dotazování nad XML, bude věnována představení současně nejpoužívanějších technologií pro dotazování nad XML. V další kapitole, XQuery algebra, budou prezentovány principy překladu dotazů na algebraické operátory. Půjde o teoretický základ pro nadefinování funkčnosti procesoru. Kapitola Implementace XQuery procesoru bude tento teoretický základ převádět do praktické podoby. Poslední kapitola, Experimentální vyhodnocování, představí naimplementovaný procesor a provede srovnání s jinými existujícími.

2 Dotazování nad XML

Tato úvodní kapitola má nastínit základní principy spojené s XML a podat stručný přehled technik, které souvisí s dotazováním.

2.1 Zamyšlení na úvod

V dnešní době se lze setkat s XML téměř ve všech oblastech informačních technologií. Nabízí se menší zamyšlení, co je důvodem tak velké popularity tak jednoduchého jazyka. Pokud se zaměříme na vývoj možností počítačového hardwaru v průběhu posledních 20-ti let a více, můžeme pozorovat obvykle exponenciální růst hodnot mnoha parametrů. Jedná se především o velikost pamětí, rychlost procesorů, přenosovou rychlost komunikačních technologií a další. Takovéto podmínky umožňují odprostit se od uvažování nad každým bajtem zvlášť a využívat vyšší úrovně abstrakce.

Pohledem např. na souborové formáty používané 15 let zpět zjistíme, že se obvykle vše ukládalo v binární podobě. Každý aplikační software měl definovaný svůj vlastní formát, jehož specifikace byla v mnoha případech z obchodních důvodů utajená. Moderní informační technologie však kladou důraz na modulární stavbu, schopnost komunikace, dodržování standardů a informační dostupnost. Individualistický přístup k řešení problémů sice může být v úzkých oblastech použití efektivnější, avšak z principu pak zabraňuje širšímu uplatnění.

Mezi nejdůležitější výhody XML patří jednoduchá, striktní, ale přesto velmi pružná syntaxe. Dále bezesporu bezplatné využívání tohoto standardu a stromová organizace, která dokáže naprosto přirozeným způsobem popsat stavbu a okamžitých stav reálných objektů. V případě rozumné volby názvů značek pro konkrétní použití, je navíc XML samopopisné, což ve většině případů umožňuje správnou interpretaci jeho obsahu i bez dostupnosti nějaké oficiální specifikace.

Konkrétně tedy XML nachází uplatnění od oblasti konfiguračních souborů, přes internetové technologie (XHTML, SOAP), souborové formáty (Open Document, Office Open XML) až po celé stromové XML databáze. Mimo to lze tento jazyk použít samozřejmě kdykoli při potřebě serializace programových objektů.

2.2 XML

e**X**tensible **M**arkup **L**anguage, čili česky rozšiřitelný značkovací jazyk, je vyvíjen a standardizován organizací W3C momentálně ve verzi 1.1 [3].

Element , tvoří základní stavební stvabení jednotku XML dokumentu. Začátek a konec elementu tvoří značky <*a*> a </*a*>, kde *a* představuje jeho název. Obsah mezi počáteční a koncovou značkou je tvořen dalšími elementy, komentáři, textem, procesními instrukcemi nebo odkazy na entity. Součástí elementu mohou být atributy, které se uvádí přímo do uvozovací značky. Je-li obsah mezi značkami prázdný, je možné vynechat ukončovací značku s tím, že je uvozovací značka doplněna o jednoduché lomítko před pravou závorkou (<*a* />).

Atribut je tvořen vždy dvojicí *klíč* ="*hodnota*". Hodnota musí být uvedena v uvozovkách, mezi sebou jsou atributy navzájem oddělovány mezerou. Atributy jsou vždy součástí elementů nebo procesních instrukcí.

Textový obsah představuje jakýkoli obecný text bez další struktury. Je nutné vyvarovat se znakům, které mohou souviset se zápisem značky, tzn. minimálně „<“ a „>“. Pro jejich vyjádření slouží tzv. znakové entity „`<`“ a „`>`“.

Komentář je zapsán mezi dvojici značek „`<!--`“ a „`-->`“. Jak bude později uvedeno, XQuery je schopen dotazovat se i na obsah komentářů, takže z hlediska zpracovávání není možné komentáře jednoduše zanedbat např. preprocesorem.

Procesní instrukce (PI) si lze představit jako speciální elementy ohraničené mezi „`<?`“ a „`?>`“. Každá PI má podobně jako element svůj název. Obsah PI je tvořen pouze atributy. Procesní instrukce se obvykle používají ke specifikování důležitých pokynů pro aplikaci, která dané XML zpracovává.

Entity umožňují předdefinovat nějaké delší opakující se úseky textu. Později se na tyto entity můžeme jednoduše odkazovat a svým způsobem tak do jisté míry eliminovat nadbytečnost dat v XML. Přesné informace o entitách je možné nalézt v [3].

Sekce CDATA slouží pro zápis surového textu. Typickým využitím je zápis úseku zdrojového kódu. Obsah CDATA se uvádí mezi značky „`<![CDATA[`“ a „`]]>`“

XML dokument je dobře formovaný (well-formed), jestliže jsou splněna všechna pravidla pro zápis výše definovaných částí. Každý XML dokument musí obsahovat přesně jeden kořenový element. Dále by se v dokumentu měla objevit procesní instrukce udávající verzi XML a použité kódování (viz. ukázka 1).

Všechny výše uvedené části XML kromě atributů bývají zobecňovány na pojem *XML uzel*.

```
<?xml version="1.0" encoding="UTF-8" ?>
<library locality="Ostrava">
  <!-- databáze knihovny v lokalitě Ostrava -->
  <section id="1">
    <book isbn="978-3-16-148410-0">
      <title>Učebnice XQuery</title>
      <author>John Black</author>
    </book>
    <book isbn="978-3-16-148411-0">
      <title>Učebnice XPath</title>
      <author>Jack White</author>
    </book>
    <book isbn="978-3-16-148412-0">
      <title>Učebnice XML</title>
      <author>Jimm Green</author>
    </book>
  </section>
</library>
```

Ukázka 1: Ukázka XML dokumentu

2.3 Dotazovací jazyky nad XML

Dotazovacích jazyků pro XML existuje v současnosti celá řada. Jednotlivé jazyky jsou více či méně používané, liší se v principech, v syntaxi, ale všechny se opírají o stromovou strukturu XML.

2.3.1 XPath

Jedná se o jednoho z prakticky nejpoužívanějších zástupců jazyků pro dotazování nad XML. Syntaxe tohoto jazyka je velmi snadná a intuitivní, takže je ho možné plně využívat už po krátkém seznámení. XPath je podobně jako vše okolo XML standardizován organizací W3C momentálně ve verzi 2.0 [4].

Filter výrazy a axis kroky

Celý XPath výraz se skládá z několika kroků dvojího typu – *axis steps* a *filter výrazů* (neplést s predikáty). Jeho vyhodnocování probíhá zleva doprava [1]. Výsledkem každého kroku je uspořádaná množina XML uzlů. Pro každý z uzlů této množiny pak proběhne zpracování kroku následujícího, kde daný uzel figuruje jako *kontextový*. *Axis step* vyjadřuje pohyb po určené ose (viz. obr. 1), *filter výraz* představuje operaci na základě kontextového XML uzlu, např. výpočet aritmetického výrazu.

Výsledná množina každého kroku může být dále filtrována pomocí predikátů v hranatých závorkách.

```
/library//book[@isbn="978-3-16-148410-0"]/author
```

Ukázka 2: Demonstrační XPath dotaz

Na ukázce 2 je zachycen jednoduchý XPath dotaz, jehož úkolem je navrátit jméno autora knihy s určitým ISBN. Dotaz bude prováděn nad dokumentem z ukázky 1.

Zápis `/library` vybere ze zdrojového dokumentu kořenový element. Pomocí `//book` dojde k vyhledání všech elementů – potomků s názvem „book“. Dále následuje zápis predikátu `[@isbn="978-3-16-148410-0"]`, který zajistí filtrování všech uzlů – knih s hodnotou atributu `isbn` rovnou „978-3-16-148410-0“. Z takto vzniklé množiny (obsahující vzhledem k dotazovanému dokumentu pouze jeden element) jsou nakonec vybráni všichni přímí potomci – uzly s názvem „author“. Uvedený výraz je ve skutečnosti zjednodušením dotazu z ukázky 3.

```
/child::library/descendant-or-self::node()/book
  [attribute::isbn="978-3-16-148410-0"]/child::author
```

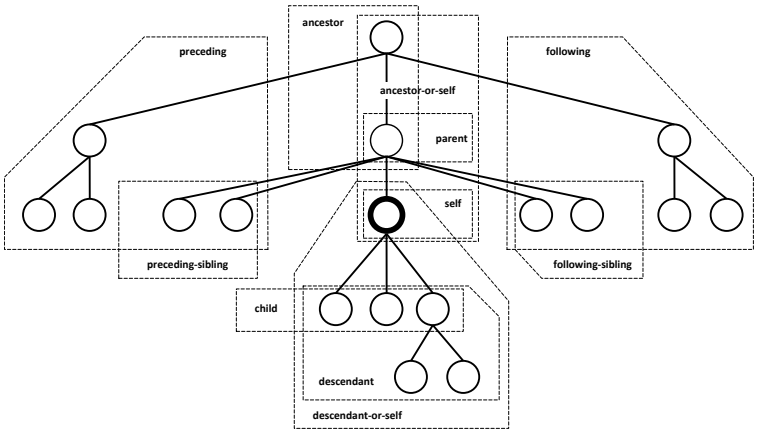
Ukázka 3: XPath dotaz bez zkratkových zápisů

[1] Vyhodnocování zleva doprava má symbolický význam pro uživatele, ve skutečnosti může procesor pohlížet na výraz jako na celek a vyhodnotit jej tak efektivněji.

Osy XML

Je vidět, že zjednodušený (zkratkový) zápis na ukázce 2 neobsahoval klíčová slova `child`, `descendant-or-self` a `attribute`. Jedná se o specifikaci tzv. os. Uvedené 3 osy jsou totiž prakticky nejpoužívanější, a tak by byl jejich plný zápis zbytečně zdlouhavý. Osy představují určení prohledávané části XML dokumentu vzhledem k uzlu, na kterém se aktuálně nacházíme, tj. kontextovému uzlu. Zápis `node()` představuje na ukázce 3 výběr libovolného XML uzlu.

Přehled os je ilustrován na obrázku 1. Aktuální kontextový uzel je zvýrazněn.



Obrázek 1: Osy XML

Zbývá podotknout, že XPath je case-sensitive, tzn. rozlišuje velikost znaků a to nejen vzhledem ke klíčovým slovům, ale také vzhledem k názvům XML uzlů.

2.3.2 XQuery

Jazyk XQuery je představitelem deklarativních programovacích jazyků. To znamená, že program, resp. v tomto případě dotaz specifikuje, co je požadováno na výstupu bez přesného udání elementárních kroků, které k výsledku povedou. Jedná se o poměrně mladý jazyk, který vznikl na základě dotazovacího jazyka Quilt [15]. Snahou XQuery je poskytnout podobný komfort při dotazování nad XML databázemi jako poskytuje SQL nad databázemi relačními.

XQuery je opět standardizován W3C, momentálně ve verzi 1.0 (od roku 2007). Standard definuje tento jazyk nejen pro tvorbu dotazů, ale také pro tvorbu celých funkcí a modulů. Součástí XQuery je možnost používání dotazů XPath. XQuery k těmto dotazům přidává některé velmi užitečné složitější konstrukce.

Aktuálně platné standardy obou jazyků – XPath a XQuery pracují se společným datovým modelem [6]. Stejně jako XPath je tento jazyk case-sensitive.

FLWOR výrazy

FLWOR výrazy [čteno jako flower] představují základní a nejvíce používanou konstrukcí XQuery. Název FLWOR vychází z prvních písmen jednotlivých *klauzulí*, tj. částí této konstrukce. Jedná se

o klauzule `for`, `let`, `where`, `order by` a `return`. Pro účely této práce budou vysvětleny pouze základní principy těchto výrazů, pro detailnější a názornější studium lze využít např. [1].

```
for $b at $i in /library//book
let $a := $b/author
where $s = "John Smith" and $i > 1
order by $b/isbn
return $b
```

Ukázka 4: Demonstrační FLWOR výraz

Význam jednotlivých klauzulí bude vysvětlen na ukázkovém dotazu 4. Jako dotazovaný dokument opět poslouží XML z ukázky 1.

Klauzule `for` říká, že bude celý zbytek výrazu iterován pro proměnnou *b* na pozici *i* přes nějakou uspořádanou množinu XML uzlů, které obdržíme XPath dotazem `/library//book`. Ve zbytku výrazu bude proměnná *b* postupně nabývat hodnot jednotlivých XML uzlů. Proměnná *i* (tzv. *poziční proměnná*) bude představovat čítač jednotlivých průchodů. Klauzulí `for` se ve FLWOR může objevit více, v tom případě půjde z hlediska uživatele o zanořené iterace. Specifikace poziční proměnné `at $i` není povinná.

Klauzule `let` slouží pro zavedení pomocné proměnné *a*, do které bude v každé iteraci `for` přiřazena hodnota XPath výrazem `$b/author`. Kromě XPath se zde mohou objevovat např. aritmetické výrazy, volání vestavěných nebo uživatelských funkcí. Klauzulí `for` a `let` se ve FLWOR může zopakovat libovolně mnoho minimálně však vždy alespoň jedna z nich.

Klauzule `where` následuje až po všech `for` a `let`. Tato klauzule specifikuje podmínku, za jakých okolností dojde skutečně k vyhodnocení dílčího výsledku následující klauzule `return`. V ukázkovém dotazu si lze všimnout použití výrazů pro porovnávání a jejich spojení logickým součinem. V případě, že podmínku specifikovat nepotřebujeme, je možné celou klauzuli `where` vynechat.

Klauzule `order by` umožňuje provést setřízení dle specifikovaných kritérií. V tomto případě jde o vzestupné abecední třízení podle ISBN jednotlivých knih, které vyhověly podmínce z klauzule `where`. XQuery umožňuje provádět i sestupné třízení klíčovým slovem `descending`, které přidáme za třídící kritérium. Kritérií pro třízení může být více, v tom případě jsou mezi sebou odděleny čárkou. Celá klauzule `order by` stejně jako `where` není povinná.

Klauzule `return` je povinnou poslední klauzulí každého FLWOR výrazu. Udáváme zde, co nakonec požadujeme na výstupu. Pomocí XPath je zde možné vracet existující XML uzly, popř. s nimi provádět nějaké řetězcové nebo aritmetické operace. Velmi často se zde objevují konstrukce nových XML uzlů.

XML konstruktory

Důležitou součástí XQuery jazyka je možnost vytvářet vlastní XML uzly, popř. celé XML stromy, jejichž obsah může být vytvářen např. pomocí FLWOR výrazů. Existují dva typy XML konstruk-

torů – *přímé (direct)* a *vypočtené (computed)*. Z praktických důvodů se obvykle používají pouze konstruktory přímé.

V případě *vypočtených konstruktorů* se XML strom vytváří pomocí speciálních klíčových slov jako `element`, `attribute`, `text`, `comment` a `processing-instruction`. Na ukázce 5 je příklad konstrukce XML elementu s názvem „a“ a textovým obsahem tvořeným obsahem proměnné *x* a textem „test“.

Konstrukce XML probíhá tak, že klíčovým slovem specifikujeme typ XML uzlu (element, atribut, textový uzel atd.), za něj (pokud to daný typ vyžaduje – elementy, atributy, procesní instrukce) umístíme jeho název a nakonec do složených závorek obsah, který se v případě elementů může skládat z dalších XML uzlů, v případě atributů z řetězcové hodnoty. Jednotlivé části obsahu oddělujeme čárkou.

```
element a { text { $x }, text { "test" } }
```

Ukázka 5: Použití vypočtených konstruktorů

Přímé konstruktory jsou na použití mnohem intuitivnější. Nové XML uzly konstruujeme velmi jednoduše tak, že je zapíšeme v jejich XML podobě, tzn. podle pravidel uvedených v kapitole 2.2. Pokud nastane situace, že v obsahu elementu nebo hodnoty atributu potřebujeme získat hodnotu nějaké proměnné např. z `let` klauzule FLWOR výrazu, umístíme do XML konstruktoru vnořený výraz do složených závorek. Stejného výsledku jako u konstruktoru z ukázky 5 dosáhneme výrazem na ukázce 6.

Přestože se může přítomnost vypočtených konstruktorů jevit jako zbytečná, není tomu tak. Vypočtené konstruktory totiž na rozdíl od přímých neumožňují specifikovat názvy uzlů pomocí proměnných ve výrazu.

```
<a>
  { $x } test
</a>
```

Ukázka 6: Použití přímých konstruktorů

Na XQuery nelze pohlížet pouze jako na filtr vstupního dokumentu. Na ukázce 7 je příklad dotazu, jehož vnitřní část pomocí přímých konstruktorů vytvoří sekvenci XML uzlů, nad kterou proběhne jednoduchý XPath. U každého výrazu, který pracuje nad XML, nezáleží na tom, zda jsou XML uzly pouze výsledkem filtrování vstupního dokumentu nebo byly vytvořeny během vyhodnocování dotazu.

```
(for $n in (<a id="1" />, <a id="2" />, <a id="3" />, <a id="4" />)
return $n)/@id
```

Ukázka 7: XQuery s konstrukcí XML

Další konstrukce XQuery

FLWOR výrazy nejsou zdaleka jedinou konstrukcí, kterou XQuery obohacuje syntaxi XPath. Minimálně za zmínku zde stojí používání alternativy, přepínače `typeswitch` nebo kvantifikovaných výrazů. Specifikace dále umožňuje tvorbu vlastních uživatelských datových typů, modulů a funkcí. Lze ukázat [11], že XQuery je Turing-kompletní jazyk.

2.3.3 XQuery Core

XQuery poskytuje velkou sadu pravidel, které umožňují jeho pohodlné používání bez nutnosti dlouhodobého studia. Z hlediska vyhodnocování dotazů je však vydefinována podmnožina tohoto jazyka známá jako *XQuery core* obsahující pouze omezené množství konstrukcí, na které lze každý XQuery dotaz přepsat. Proces přepisu dotazu XQuery na XQuery Core bývá nazýván jako tzv. *normalizace*. Normalizace vstupního dotazu je běžným počátečním krokem vyhodnocování u většiny procesorů.

Ukázka 8 představuje normalizovanou podobu jednoduchého XPath výrazu `$a/book/isbn`. Z uvedené ukázky je patrné, že z relativně jednoduchého výrazu vznikla poměrně složitá konstrukce skládající se ze dvou FLWOR a pomocné funkce `ddo`, která eliminuje duplicitní výskyty uzlů a řadí uzly podle původního pořadí v dokumentu.

```
ddo(
  let $seq := ddo($a)
  let $last := fn:count($seq)
  for $dot at $position in $seq
  return
    let $seq := ddo(child::book)
    let $last := fn:count($seq)
    for $dot at $position in $seq
    return child::isbn)
```

Ukázka 8: Normalizace XPath výrazu `$a/book/isbn`

XQuery Core např. nepodporuje aritmetické výrazy, výrazy pro porovnávání, logické spojky – to vše se obvykle nahrazuje voláním vestavěných funkcí, dále nejsou definovány výrazy XPath, přímé konstruktory nebo klauzule `order by` FLWOR výrazů. Normalizace je výborným nástrojem pro specifikaci sémantiky XQuery [7], avšak ne vždy je vhodné provést normalizaci přesně dle doporučení W3C, jelikož se tak v určitých případech můžeme připravit o efektivnější variantu vyhodnocování.

Původní zadání práce se omezuje pouze na překlad normalizovaných XQuery dotazů. Výsledkem je však procesor, který si poradí i s mnoha dotazy, které normalizací neprošly. Jsou tedy podporovány přímé konstruktory, XPath výrazy a další konstrukce, které XQuery Core nedefinuje (viz. kapitola 4.1).

2.3.4 XSLT

Posledním jazykem, který zde bude spíše pro úplnost zmíněn je XSLT (e**X**tensible **S**tylesheet **L**anguage **T**ransformation). Nejedná se o dotazovací jazyk v pravém slova smyslu, XSLT je primárně navržen k provádění transformací XML dokumentů např. do prezentovatelné podoby ve formě HTML. V kombinaci s jazykem XSL Formatting Objects je pak možné provádět např. exporty do formátu PDF.

Podobně jako u XQuery tvoří XPath součást jazyka tohoto jazyka. Je zde opět použitý stejný datový model [6]. Podrobnější informace o XSLT včetně ukázkových tutoriálů lze nalézt v [1].

Vzhledem k tomu, že se všechny tři uvedené jazyky (XPath, XQuery a XSLT) často překrývají (minimálně v datovém modelu), je běžnou praxí, že dostupné implementace (např. Saxon [10] nebo XML Prime [2]) nejsou zaměřené pouze na jeden jediný jazyk, ale poskytují komplexní řešení od zpracování XML přes více různých způsobů dotazování.

3 XQuery algebra

Nejen v případě XQuery, ale také např. jazyka SQL, neprobíhá vyhodnocování vstupního dotazu interpretací. Dotaz je obvykle nejprve zkompilován na strom operátorů, které jsou později v přesně určeném pořadí vyhodnocovány. Co to operátory jsou a jakým způsobem pracují popisuje právě algebra. Algeber existuje celá řada, liší se jak svým účelem (relační databáze, stromové databáze), tak množinou definovaných operátorů a datovým modelem. V souvislosti s XQuery se obvykle používá algebra založená na n-ticovém datovém modelu. Algebra zde popsaná a později implementovaná vychází z článku [16] a kombinuje vyhodnocování na základě toku n-tic s vyhodnocováním na základě toku datových položek.

3.1 Datový model

Před vysvětlením samotné algebry je nutné nejprve ustanovit datový model a nadefinovat vhodné formální značení, které pak bude používáno k popisu funkčnosti jednotlivých operátorů. Později v implementačních detailech bude tento model přesněji specifikován na třídním diagramu, který vymezí základní datové typy celého procesoru (viz. kapitola 4.4).

3.1.1 Položka (item)

Položkou může být libovolná atomická hodnota, tzn. číslo, textový řetězec nebo hodnota typu boolean (true / false). Za položku se rovněž považuje jakýkoli XML uzel, tzn. element, atribut, textový uzel nebo komentář. Entity, sekce CDATA a procesní instrukce jsou zde pro zjednodušení vynechány. Položky budou formálně značeny jako i_k. Ve speciálním případě, pokud půjde o atomické hodnoty pak a_k.

3.1.2 Sekvence (sequence)

Sekvencí se rozumí uspořádaná multimnožina položek. Formálně bude sekvence značena jako $S = \langle i_1, i_2, \ldots, i_n \rangle$, kde i_k $(1 \leq k \leq n)$ je položka sekvence S a n značí délku této sekvence. Prázdná sekvence, tj. sekvence bez položek, je značena jednoduše $\langle\rangle$. V další části textu bude možné pohlížet na jednoprvkové sekvence jako na položky a naopak.

3.1.3 N-tice (tuple)

N-tice představuje n-rozměrný vektor s jednoznačně pojmenovanými složkami, které jsou tvořeny výše definovanými sekvencemi. V terminologii relačních databází je ekvivalentem n-tice záznam. Formální zápis n-tice bude vypadat jako $\tau = [q_1 : S_1, q_2 : S_2, \ldots, q_n : S_n]$, kde $q_1, q_2, \ldots, q_n$ představují označení (názvy) složek a $S_1, S_2, \ldots, S_n$ jednotlivé sekvence. Opět v návaznosti na relační databáze bychom mohli říci, že $q_1, q_2, \ldots, q_n$ představují názvy atributů. V případě, že $n = 0$ obsahuje n-tice nulový počet složek a jde o tzv. prázdnou n-tici, kterou zapisujeme přirozeně jako [].

3.1.4 Schéma n-tice

Z předchozí definice vyplývá nutnost rozlišovat n-tice s různou strukturou, tzn. s různým značením $q_1, q_2, \ldots, q_n$. Takové značení bude nazýváno schématem $Q = \langle q_1, q_2, \ldots, q_n \rangle$.

3.1.5 Tabulka (table)

Stejně tak jako položky mohly vytvářet sekvence, mohou n-tice vytvářet tabulky. Tabulka tedy představuje uspořádanou multimnožinu n-tic a formálně ji značíme jako $\sigma = \langle \tau_1, \tau_2, \ldots, \tau_n \rangle$. Uvedená definice tabulky obecně umožňuje sdružovat libovolné n-tice, tzn. i s různým schématem. Nás ale budou zajímat především tzv. homogenní tabulky, kde budou mít všechny n-tice stejné schéma.

3.2 Operátory

Operátorem se obecně rozumí nějaká funkce, která transformuje jeden nebo více vstupů podle určitých pravidel na výstup. V případě např. základních aritmetických operátorů (+, −, ∗, :) je situace poměrně snadná – máme dva nezávislé vstupy (levý a pravý operand), na základě kterých operátor spočítá výstup.

V případě XQuery (a nejen XQuery) je ale situace trochu komplikovanější. Konečný výsledek totiž může záviset nejen na nezávislých vstupech, ale také na nějakém předem neznámém výsledku mezivýpočtu. Typickým příkladem je běžný operátor relační algebry – selekce, kterému je potřeba kromě nezávislé vstupní množiny záznamů specifikovat také podmínku, za jakých okolností bude zpracovávaný záznam součástí výstupu.

Obecná struktura operátoru vypadá následovně:

$$Op[q_1, \ldots, q_k]\{DOp_1, \ldots, DOp_l\}(Op_1, \ldots, Op_m)$$

$q_1, \ldots, q_k$ jsou statické složky operátoru. Může se jednat např. o názvy proměnných, specifikaci datového typu nebo specifikaci názvu XML elementu. Podmínkou je, aby byly tyto hodnoty známé ještě před začátkem vyhodnocování.

$DOp_1, \ldots, DOp_l$ jsou tzv. závislé operátory (dependent). Jedná se právě o ty operátory, které slouží pro výpočet mezivýsledku.

$Op_1, \ldots, Op_m$ jsou nezávislé operátory – vstupy. Najdeme je téměř ve všech operátorech. Výjimku tvoří pouze ty, které vracejí nějakou staticky danou skalární hodnotu nebo přistupují ke kontextovým proměnným. Jak závislé, tak nezávislé operátory jsou vždy operátory. To znamená, že se na místě operátoru nikdy nevyskytne žádná skalární hodnota. Pro vytvoření skalární hodnoty existuje speciální operátor, který má skutečnou skalární hodnotu danou pomocí statické složky (viz. kapitola 3.3.1).

3.2.1 Strom operátorů

Vzhledem k tomu, že se operátor obvykle skládá z dalších závislých nebo nezávislých suboperátorů, vznikne kompilací hierarchie reprezentovatelná stromem, kde každý operátor představuje jeden vrchol a hrana odkaz na suboperátor. Operátory bez závislých a nezávislých suboperátorů utvoří listy. Celý strom operátorů, který je výsledkem kompilace, bývá nazýván také jako *plán vykonávání dotazu*. Ukázkový strom operátorů je možné vidět např. na obrázku 2 níže.

3.2.2 Výpočet

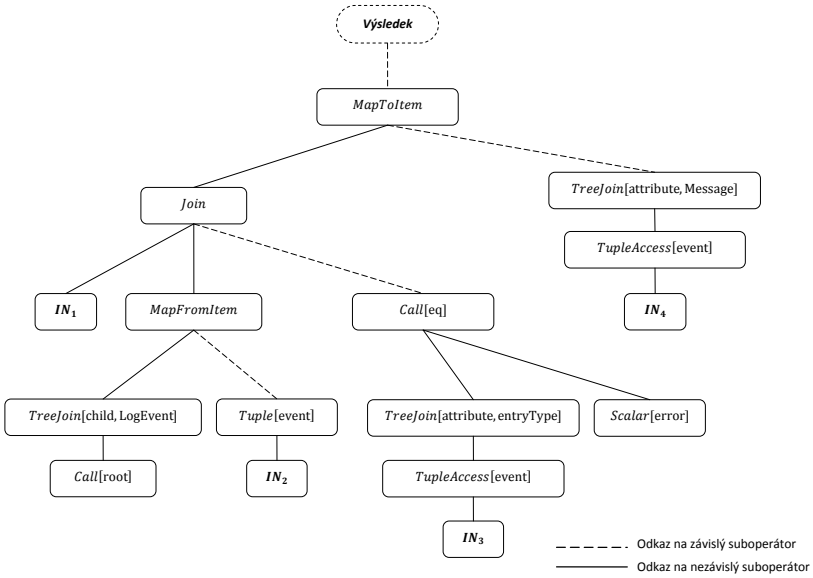
Plán vykonávání dotazu udává pořadí v jakém budou jednotlivé operátory vyhodnocovány. Výpočet vždy začíná a končí *kořenovým operátorem*, tzn. tím, který tvoří kořen stromu. Každý operátor nejprve vyhodnotí všechny své nezávislé vstupy a až poté na jejich základě případně vyhodnotí vstupy závislé, čili mezivýsledky. Podle výsledků nezávislých vstupů, dílčích mezivýsledků nebo případně statických složek provede svou vlastní úlohu a vrátí výsledek. Výsledek kořenového operátoru je výsledkem vyhodnocování celého dotazu.

3.2.3 Operátor IN

Každý operátor je vyhodnocován v rámci nějaké kontextové hodnoty – n-tice nebo položky. Tuto kontextovou hodnotu (zkráceně kontext) nastaví nějaký operátor *Op* se závislým suboperátorem Op_{sub} při požadavku na jeho vyhodnocení. Obsahuje-li *Op* vstupy nezávislé, pak jsou tyto vyhodnoceny se stejným kontextem jako *Op*. Účelem operátoru IN je pouhé navrácení tohoto kontextu.

Příklad 1: *Význam operátoru* IN

V ukázkovém stromu na obrázku 2 je IN použitý celkem 4 krát. Pro lepší orientaci jsou jeho výskyty rozlišeny spodním indexem. Co přesně které operátory znamenají není momentálně důležité. Podstatné je uvědomit si, který operátor nastavil kterou kontextovou hodnotu, jež získáváme operátorem IN. To je možné zjistit jednoduše tak, že budeme procházet strom od konkrétního listu IN postupně na rodiče tak dlouho, než projdeme hranou představující odkaz na závislý suboperátor.



Obrázek 2: Ukázka stromu operátorů

Pro IN_2 je to tedy operátor MapFromItem, pro IN_3 operátor Join a pro IN_4 operátor MapToItem. IN_1 je jediný případ, který nesouvisí s žádným operátorem se závislým vstupem. Je možné si představit, že kořenový operátor MapToItem je vyhodnocován jako závislý vstup vzhledem k nějaké programové funkci, která zahajuje výpočet výsledku dotazu s nějakým výchozím kontextem n-ticí, jejíž obsahem mohou být např. globální proměnné nebo kontextový XML uzel nastavený na dotazovaný XML dokument.

V tuto chvíli spíše pro zajímavost – strom na obr. 2 je plánem pro XQuery dotaz na ukázce 9.

```
for $event in /EventLogeEvent
where $event/@Entry_Type = "Error"
return $event/@Message
```

Ukázka 9: Ukázkový XQuery dotaz

Kompletní seznam operátorů je k dispozici v tabulce 1, která je s malými úpravami přebrána z článku [16], tabulky 1. Oproti [16] jsou zde vynechány nepoužité operátory, naopak se zde navíc vyskytují And, Or a IN. Funkce některých méně známých důležitých operátorů bude vysvětlena později při popisu kompilačních pravidel.

3.3 Kompilační pravidla

Tato podkapitola je věnována problematice, jak seskládat strom operátorů tak, aby jeho vyhodnocení vedlo ke korektnímu výsledku. Nejprve budou popsány elementární kompilační kroky, na nichž bude ilustrována funkčnost jednoduchých operátorů. Později pak bude rozebrán překlad složitějších konstrukcí – FLWOR a XPath výrazů.

Pro formální popis sémantiky kompilačních pravidel bude použita následující notace:

$$\frac{\begin{array}{c}.\\.\\.\end{array}}{[\![expr]\!]_{contextOp} \Rightarrow Op_{out}}$$

Tato notace představuje překlad výrazu *expr* na strom operátorů s kořenem Op_{out} v kontextu operátoru *contextOp* (pozor, neplést s kontextovou hodnotou) podle pravidel uvedených nad čarou. Kontextový operátor *contextOp* se používá pouze při popisování kompilace FLWOR a XPath výrazů. Kompilace těchto konstrukcí je rozdělena na několik částí, kde výsledný operátor překladu jedné části vstupuje jako kontextový operátor do překladu části následující.

Pravidla nad čarou jsou dvojího typu. Zápis $expr \Rightarrow Op$ znamená dílčí překlad výrazu *expr* na operátor *Op*. O tom, jaká kompilační pravidla se použijí rozhodne až konkrétní podoba výrazu *expr*.

Druhým pravidlem je prosté přiřazení ve tvaru *Op* = (rozklad na operátory). Jedná se o dále nedělitelný kompilační krok, který představuje konstrukci podstromu operátorů podle pravé strany a přiřazení do operátoru na levé straně.

Operátory pracující nad položkami			
Sekvence	$\text{Sequence}(S_1, S_2, \ldots, S_n)$	$\rightarrow$	S_{out}
Prázdná sekvence	$\text{Empty}()$	$\rightarrow$	S_{out}
Skalární hodnota	$\text{Scalar}[a]()$	$\rightarrow$	a
XML element	$\text{Element}[q](S_{in})$	$\rightarrow$	`element q {`S_{in}`}`
XML atribut	$\text{Attribute}[q](a)$	$\rightarrow$	`attribute q {`a`}`
XML text	$\text{Text}(a)$	$\rightarrow$	`text q {`a`}`
XML komentář	$\text{Comment}(a)$	$\rightarrow$	`comment q {`a`}`
Tree join	$\text{TreeJoin}[axis, nodetest](S_{in})$	$\rightarrow$	S_{out}
Test přetypovatelnosti	$\text{Castable}[type](a)$	$\rightarrow$	*boolean*
Přetypování	$\text{Cast}[type](a_{in})$	$\rightarrow$	a_{out}
Kontrola datového typu	$\text{TypeMatches}[type](S_{in})$	$\rightarrow$	*boolean*
Přístup k proměnné	$\text{Var}[q]$	$\rightarrow$	a
Volání funkce	$\text{Call}[q](S_1, S_2, \ldots, S_n)$	$\rightarrow$	S_{out}
Alternativa	$\text{Cond}\{S_a, S_b\}(boolean)$	$\rightarrow$	S_{out}
Logický součet	$\text{Or}\{b_1, b_2, \ldots, b_n\}()$	$\rightarrow$	*boolean*
Logický součin	$\text{And}\{b_1, b_2, \ldots, b_n\}()$	$\rightarrow$	*boolean*
Operátory pracující nad n-ticemi			
Konstrukce n-tice	$\text{Tuple}[q_1, \ldots, q_n](S_1, \ldots, S_n)$	$\rightarrow$	$[q_1 : S_1, \ldots, q_n : S_n]$
Spojení n-tice	$\text{TupleConcat}(\tau_1, \tau_2)$	$\rightarrow$	τ_{out}
Přístup ke složce	$\text{TupleAccess}[q](\tau)$	$\rightarrow$	i_{out}
Selekce	$\text{Select}\{boolean\}(\sigma_{in})$	$\rightarrow$	σ_{out}
Kartézský součin	$\text{Product}(\sigma_1, \sigma_2)$	$\rightarrow$	σ_{out}
Spojení	$\text{Join}\{boolean\}(\sigma_1, \sigma_2)$	$\rightarrow$	σ_{out}
Závislé spojení	$\text{MapConcat}\{boolean\}(\sigma_1, \sigma_2)$	$\rightarrow$	σ_{out}
Mapování indexu	$\text{MapIndex}[q](\sigma_{in})$	$\rightarrow$	σ_{out}
Třídění	$\text{OrderBy}[q_1, \ldots, q_n, mod_1, \ldots, mod_n](\sigma_{in})$	$\rightarrow$	σ_{out}
Hybridní operátory			
Převod sekvence na tabulku	$\text{MapFromItem}\{\tau\}(S_{in})$	$\rightarrow$	σ_{out}
Převod tabulky na sekvenci	$\text{MapToItem}\{i\}(\sigma_{in})$	$\rightarrow$	S_{out}
Existenční kvantifikátor	$\text{MapSome}\{boolean\}(S_{in})$	$\rightarrow$	*boolean*
Všeobecný kvantifikátor	$\text{MapEvery}\{boolean\}(S_{in})$	$\rightarrow$	*boolean*
Speciální operátory			
Navrácení kontextu	$\text{IN}()$	$\rightarrow$	τ_{out} nebo $\imath_{out}$

Tabulka 1: Operátory algebry

Pravidla jsou aplikována přesně v takovém pořadí, jak jdou za sebou. Je potřeba si uvědomit, že provádíme kompilaci, ne vyhodnocování. To znamená, že výpočet teprve připravujeme, ale neprovádíme.

3.3.1 Kompilace elementárních konstrukcí

Položka

$$\frac{Op_{out} = \text{Scalar}[a]}{a \Rightarrow Op_{out}}$$

Výsledkem kompilace je vždy strom operátorů. Jak již bylo uvedeno, v tomto stromu se nikdy ani na straně závislých, ani na straně nezávislých operátorů nevyskytuje skalární hodnota. Jakoukoli skalární hodnotu je potřeba nejprve zkonstruovat operátorem Scalar, kterému skutečnou požadovanou hodnotu předáme jako statický parametr.

Proměnná

$$\frac{Op_{out} = \text{Var}[v]}{\$v \Rightarrow Op_{out}}$$

Přístup k proměnné v jednoduše kompilujeme na operátor Var[v]. Jak bude ale později vysvětleno, prakticky se bude tento operátor vždy nahrazovat přístupem k v složce kontextové n-tice.

Sekvence

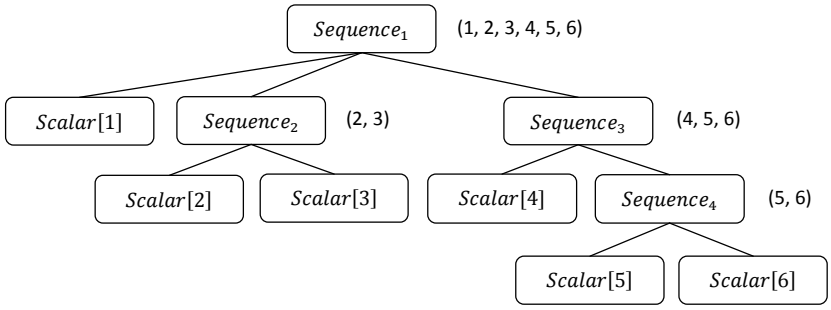
$$\frac{\begin{array}{c} expr_1 \Rightarrow Op_1 \\ \vdots \\ expr_n \Rightarrow Op_n \\ Op_{out} = \text{Sequence}(Op_1, \ldots, Op_n) \end{array}}{(expr_1, \ldots, expr_n) \Rightarrow Op_{out}}$$

Jednotlivé výrazy $expr_1, \ldots, expr_n$ nemusí ve výsledku dávat pouze položky. Všechny dílčí sekvence spojí operátor Sequence do jedné výsledné (viz. obrázek 3). Znamená to, že například zápis `(1,(2,3),(4,(5,6)))` znamená totéž co `(1,2,3,4,5,6)`. Datový model neumožňuje, aby položka obsahovala vnořenou sekvenci.

Funkce

$$\frac{\begin{array}{c} arg_1 \Rightarrow Op_1 \\ \vdots \\ arg_n \Rightarrow Op_n \\ Op_{out} = Call[f](Op_1, \ldots, Op_n) \end{array}}{f(arg_1, \ldots, arg_n) \Rightarrow Op_{out}}$$

Při kompilaci přechází volání funkce přirozeně na operátor *Call*. Tento operátor má statickým parametrem určen název funkce f a obecně n nezávislých vstupů určuje hodnoty argumentů.



Obrázek 3: Kompilace sekvence

Aritmetické výrazy

Do této chvíle možná nemusí být úplně jasné, kam se vlastně poděly aritmetické operátory. V tabulce 1 byly nadefinovány nejrůznější speciální operátory, ale žádné z nich nepřipomínají např. operátor součtu, rozdílu atd. Určitě to není tím, že by snad XQuery takové výrazy nepodporoval. Malý trik spočívá v tom, že lze veškeré tyto operace nahradit voláním vestavěných funkcí. Při kompilaci součtu, rozdílu, součinu, podílu nebo operace modulo se tedy jednoduše zavolají dvouparametrové funkce `add`, `sub`, `mul`, `div` nebo `mod`. O nahrazení aritmetických operátorů voláním vestavěných funkcí se běžně stará normalizace (viz. kapitola 2.3.3) – XQuery Core skutečně aritmetické výrazy nepodporuje.

Konkrétně např. pro operaci součtu bude překlad vypadat takto:

$$\frac{\begin{array}{c} expr_a \Rightarrow Op_a \\ expr_b \Rightarrow Op_b \\ Op_{out} = \text{Call[add]}(Op_a, Op_b) \end{array}}{expr_a + expr_b \Rightarrow Op_{out}}$$

Levá i pravá strana součtu bude přeložena na pomocné operátory Op_a a Op_b, které pak nastavíme na vstup operátoru Call se staticky daným názvem funkce `add`. Unární minus lze jednoduše zkompilovat stejně jako binární variantu s tím, že na levé straně bude 0 (po kompilaci na operátor Scalar).

Výrazy porovnávání

XQuery jazyk rozlišuje celkem 4 typy porovnávání – *obecné porovnání*, *hodnotové porovnání*, *porovnání Xml uzlu* a *porovnání pořadí Xml uzlu*.

Nejčastěji je možné setkat se s obecným porovnáním, kde srovnáváme obsah dvou sekvencí S_1 a S_2. Jde o standardní zápis symbolů `=`, `!=`, `<`, `>`, `<=` a `>=`, které se při kompilaci přeloží na volání dvouargumentových funkcí `eq`, `neq`, `lt`, `gt`, `le` a `ge`. Tyto funkce vracejí pravdivou booleovskou hodnotu za předpokladu, že zadaná rovnost nebo nerovnost vyhoví pro alespoň jednu dvojici položek $i_1 \in S_1$ a $i_2 \in S_2$. Ve většině případů použití jsou S_1 a S_2 jednoprvkové.

Druhým typem porovnání je tzv. hodnotové, kdy nesrovnáváme dvě sekvence, ale výhradně

dvě položky. Pro tato porovnání má XQuery speciální klíčová slova `eq`, `neq`, `lt`, `gt`, `le` a `ge`, které se používají naprosto stejně jako standardní symboly u obecného porovnání. Při kompilaci se hodnotové porovnání přeloží na volání funkcí `veq`, `vneq`, `vlt`, `vgt`, `vle` a `vge`.

Další dva uvedené typy porovnání nejsou v této práci implementovány, avšak jejich překlad by proběhl velice podobně opět na volání funkcí.

Logické výrazy

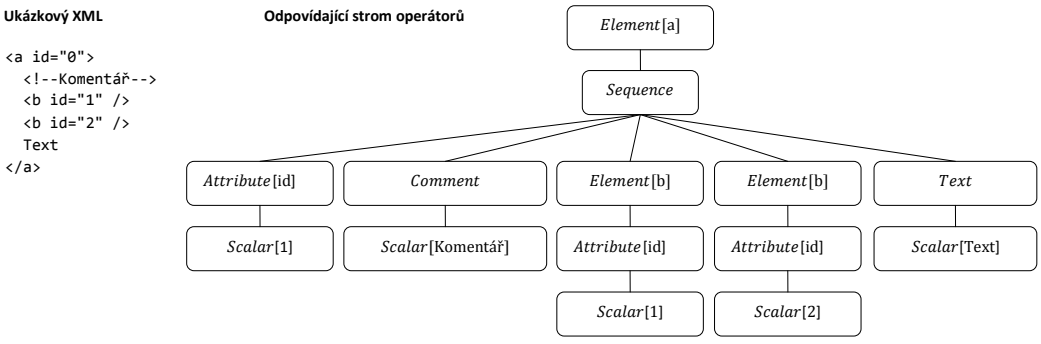
Logické operace (logický součin a součet) by sice bylo možné podobným způsobem přepsat na funkce `and` a `or`, nicméně přišli bychom tak o podstatnou výhodu možnosti líného vyhodnocování. Při zpracovávání operátoru Call se chtě nechtě nejprve musí vyhodnotit všechny vstupy, což je u logických výrazů obvykle zbytečné. Proto zde přeci jen použijeme speciální operátory Or a And, které jednotlivé vstupy vyhodnocují postupně. Jakmile logický součet narazí na „true“ nebo logický součin na „false“, dojde k okamžitému navrácení výsledku.

$$\frac{\begin{array}{c} arg_1 \Rightarrow Op_1 \\ \vdots \\ arg_n \Rightarrow Op_n \\ Op_{out} = \mathrm{And}\{Op_1, \ldots, Op_n\}() \end{array}}{arg_1 \text{ and } \ldots \text{ and } arg_n \Rightarrow Op_{out}} \qquad \frac{\begin{array}{c} arg_1 \Rightarrow Op_1 \\ \vdots \\ arg_n \Rightarrow Op_n \\ Op_{out} = \mathrm{Or}\{Op_1, \ldots, Op_n\}() \end{array}}{arg_1 \texttt{ or } \ldots \texttt{ or } arg_n \Rightarrow Op_{out}}$$

XML konstruktory

Jak bylo v kapitole 2.3.2 uvedeno, v dotazech XQuery můžeme pomocí přímých nebo nepřímých konstruktorů vytvářet vlastní XML uzly.

Na obrázek 4 lze pohlížet dvěma způsoby – buď jako na sestavení stromu operátorů podle vstupního dotazu nebo jako na výsledek vyhodnocení uvedeného stromu. Jelikož v dotazu nejsou použité žádné charakteristické konstrukce pro XQuery, bude vyhodnocování spočívat pouze v parsování XML na DOM a zpětné serializaci DOM na textový řetězec.



Obrázek 4: Kompilace XML

3.3.2 FLWOR výrazy

Kompilační pravidla budou aplikována postupně přesně v takovém pořadí, v jakém jdou za sebou jednotlivé klauzule. Gramatika XQuery umožňuje několikrát zopakovat klauzuli `for` a `let` v libovolném pořadí, následuje nepovinná omezující podmínka `where`, nepovinné setřízení `order by` a povinná klauzule `return`. Celý FLWOR výraz bude zpracováván pomocí operátorů pracujících nad tabulkami, pouze v případě vstupního výrazu nebo výpočtu výsledku potřebovat ony tzv. hybridní operátory. Pravidla pro překlad jsou s malými úpravami převzata z [16].

for

Bude iterována proměnná x přes vstupní sekvenci položek danou výrazem *expr*. Vstupní sekvence položek může být staticky zadána nebo může vycházet například z výsledku vnořeného XPath nebo FLWOR výrazu.

$$\frac{\begin{array}{c} expr \Rightarrow Op_1 \\ [\![\texttt{as } T]\!]_{Op_1} \Rightarrow Op_1 \\ Op_2 = \text{MapFromItem}\{Tuple[x](\text{IN})\}(Op_1) \\ Op_2 = \text{MapIndex}[pos](Op_2) \\ Op_3 = \text{MapConcat}\{Op_2\}(Op_0) \\ [\![clauses]\!]_{Op_3} \Rightarrow Op_{out} \end{array}}{[\![\texttt{for } \$x \texttt{ [as } T\texttt{] [at } \$pos\texttt{] in } expr\ clauses]\!]_{Op_0} \Rightarrow Op_{out}}$$

V prvním kroku je zkompilován výraz *expr* na pomocný operátor Op_1. Nejedná se samozřejmě o jeden operátor, ale o celý podstrom s kořenem Op_1. Druhý krok je nepovinný. Provádí se v případě nutnosti ošetření datového typu vstupní sekvence. Klauzuli `as` tato implementace kvůli zjednodušenému typovému systému nepodporuje (viz. kapitola 4.4.1).

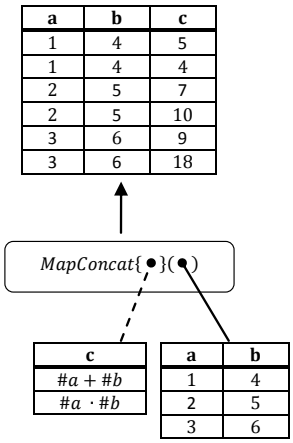
Ve třetím kroku je proveden přechod od sekvence položek k jednoatributové tabulce, tzn. k tabulce obsahující n-tice s jedinou složkou x. Čtvrtý krok je opět nepovinný. Záleží na tom, zda hodláme využít možnosti poziční proměnné `at $pos`. Pokud ano, pak operátorem MapIndex do vznikající tabulky jednoduše přidáme složku *pos* s celočíselnou hodnotou v rozmezí $\{1..|expr|\}$, tzn. zajistíme očíslování jednotlivých n-tic. V zápise kompilačního pravidla je v tomto kroku případně přepsán pomocný operátor Op_2.

Pátý krok využívá operátor MapConcat. Ten zajistí, že prozatím připravený výpočet v operátoru Op_2 bude proveden pro každou n-tici z výsledku operátoru, který vznikl kompilací předchozí `for` nebo `let` klauzule. Princip činnosti MapConcat je vysvětlen níže.

Poslední krok překladu `for` nepředstavuje nic jiného, než zpracování zbytku FLWOR výrazu s tím, že operátor Op_3 bude dalšímu kroku předán jako kontextový. Je zde jedna drobná komplikace, na kterou nesmíme zapomenout. Ve zpracování zbytku FLWOR výrazu musí být jakýkoli přístup k proměnné x kompilován ne na operátor Var[x], ale na TupleAccess[x](IN), tj. přístup ke složce x kontextové n-tice. Stejné pravidlo bude případně platit také pro poziční proměnnou *pos*. Substituce probíhá z toho důvodu, že ve skutečnosti neexistuje žádná paměťová struktura, která by si pamatovala hodnoty proměnných samostatně. Všechny hodnoty jsou obsaženy ve vznikající tabulce n-tic.

Operátor MapConcat může být obtížnější na správné pochopení. Na obrázku 5 je znázorněn princip jeho činnosti. Tento operátor přebírá nezávislým vstupem tabulku. Pro každou n-tici ze

vstupní tabulky vyhodnotí závislý operátor, který v tomto případě produkuje další tabulku o 2 n-ticích. N-tice ze vstupní tabulky je ve výpočtu závislého suboperátoru dostupná v kontextu. Zápis *#a* je zkrácením přístupu ke složce *a* kontextové n-tice. Z ukázky je patrné, že činnost MapConcat je velmi podobná kartézskému součinu (operátor Product), obsah pravé připojované tabulky však závisí na obsahu levé nezávislé tabulky.



Obrázek 5: Činnost operátoru MapConcat

let

Klauzule `let` zavádí do FLWOR výrazu závislou proměnnou y. Úkolem operátorů je v tomto případě přidat do vznikající tabulky složku y s hodnotou vypočtenou podle *expr*.

$$\frac{\begin{array}{c} expr \Rightarrow Op_1 \\ [\![\texttt{as } T]\!]_{Op_1} \Rightarrow Op_1 \\ Op_2 = \text{MapConcat}\{\text{Tuple}[y](Op_1)\}(Op_0) \\ [\![clauses]\!]_{Op_3} \Rightarrow Op_{out} \end{array}}{[\![\texttt{let \$}y\texttt{ [as } T\texttt{]} := expr\ clauses]\!]_{Op_0} \Rightarrow Op_{out}}$$

V prvním kroku je přeložen výraz *expr* do operátoru Op_1. Druhý krok opět představuje ošetření datového typu, kterým se zabývat nebudeme. Pomocí MapConcat dojde k připojení jednosložkové n-tice $[y : expr]$ k tabulce vzniklé vyhodnocením kontextového operátoru Op_0, tzn. tabulce, která je výsledkem všech předchozích klauzulí `for` a `let` překládaného FLWOR výrazu. Nakonec opět pokračujeme zpracováním zbytku, kde stejně jako v případě `for` budeme nahrazovat jakýkoli výskyt proměnné y přístupem ke složce y kontextové n-tice.

V čem se tedy zásadně liší zpracování klauzulí `for` a `let`? Rozdíl je ve zpracování vstupního výrazu `expr`. Zatímco u `for` převádíme sekvenci danou *expr* na tabulku s n-ticemi obsahujícími jednotlivé dílčí položky sekvence, u klauzule `let` necháváme sekvenci v celku a tvoříme jedinou n-tici se složkou y, která tuto sekvenci obsahuje. Z klauzule `let` je patrné, proč datový model umožňuje složkám n-tic obsahovat celé sekvence.

where

Zatímco předchozí `for` a `let` sloužily k vytvoření tabulky, `where` bude vzniklou tabulku podle dané podmínky pouze redukovat. Základem zpracování této klauzule je všeobecně známý operátor relační algebry Select.

$$\frac{\begin{array}{c} expr \Rightarrow Op_1 \\ Op_2 = \text{Select}\{Op_1\}(Op_0) \\ [\![clauses]\!]_{Op_2} \Rightarrow Op_{out} \end{array}}{[\![\texttt{where}\ expr\ clauses]\!]_{Op_0} \Rightarrow Op_{out}}$$

Výraz *expr* je tedy zkompilován na pomocný operátor Op_1. Kontextový operátor Op_0, čili ten, který produkuje tabulku podle předchozích `let` a `for`, bude sloužit jako nezávislý vstup do operátoru Select, závislým vstupem selekce bude podmínka zkompilovaná v Op_1. Pokračujeme další klauzulí a jelikož nedošlo k žádnému zavádění proměnné, není potřeba ošetřovat substituci.

order by

$$\frac{\begin{array}{c} expr_1 \Rightarrow Op_1 \\ \vdots \\ expr_n \Rightarrow Op_n \\ Op_0 = \text{MapConcat}\{\text{Tuple}[o_1, \ldots, o_n](Op_1, \ldots, Op_n)\}(Op_0) \\ Op_0 = \text{OrderBy}[o_1, \ldots, o_n, modifier_1, \ldots, modifier_n](Op_0) \\ [\![clauses]\!]_{Op_0} \Rightarrow Op_{out} \end{array}}{[\![\texttt{order by}\ expr_1\ modifier_1, \ldots, expr_n\ modifier_n]\!]_{Op_0} \Rightarrow Op_{out}}$$

Nejprve budou zkompilovány všechny výrazy $expr_1 \ldots expr_n$, podle kterých budeme provádět třídění, na operátory $Op_1 \ldots Op_n$. Poté dojde k připojení n-tic se složkami jednotlivých vypočtených výrazů k výsledku Op_0. Teprve pak je možné aplikovat operátor OrderBy, kterému staticky nadefinujeme, podle kterých složek n-tic v tabulce má třídit a jakým způsobem, tzn. vzestupně (`ascending`), sestupně (`descending`), popř. jak má být tříděna prázdná sekvence (`empty least` nebo `empty greatest`). Opět pokračujeme zpracováváním zbytku FLWOR výrazu, kterým je v tuto chvíli jistě pouze klauzule `return`.

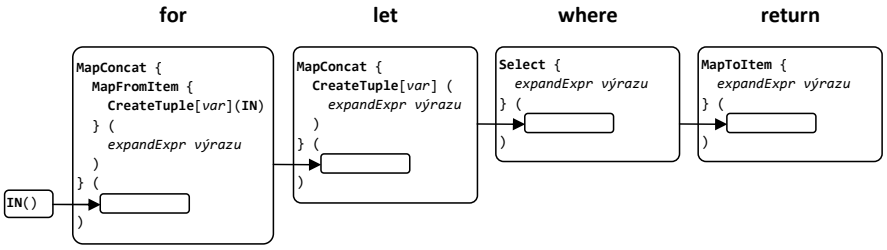
Drobnou komplikací je pouze to, že zde do tabulky mapujeme složky n-tic $o_1, \ldots, o_n$, o které si sám uživatel nežádal. Je potřeba dát pozor, abychom se nevhodnou volbou pojmenování složky n-tice, nedostali do konfliktu s nějakou uživatelem definovanou proměnnou.

return

$$\frac{\begin{array}{c} expr \Rightarrow Op_1 \\ Op_{out} = \text{MapToItem}\{Op_1\}(Op_0) \end{array}}{[\![\texttt{return}\ expr]\!]_{Op_0} \Rightarrow Op_{out}}$$

Poslední krok překladu FLWOR je v celku jednoduchý. Nejprve dojde ke kompilaci výrazu, který představuje výpočet jedné dílčí položky výsledku. Pomocí operátoru MapToItem je zajištěno, že tento výraz bude vyhodnocen pro každou n-tici z výsledku kontextového operátoru Op_0. Všechny dílčí výsledky nakonec utvoří výslednou sekvenci.

Na obrázku 6 je v blokovém schématu shrnuto, jakým způsobem probíhá překlad jednotlivých klauzulí podle výše uvedených pravidel. Pro zjednodušení je vynechána klauzule `order by`. Během celé kompilace FLWOR se pracuje s proměnnou kontextového operátoru. Každá klauzule tuto proměnnou zaobalí jiným operátorem nebo více operátory.



Obrázek 6: Blokové schéma kompilace FLWOR

Následuje souhrnná ukázka, která demonstruje výsledek většiny výše nadefinovaných kompilačních pravidel.

Příklad 2: *Souhrnný příklad kompilace*

Pokusme se nyní přeložit následující výraz:

```
for $i in 1 to 10
let $j := $i * 2
where $j > 10
return $i
```

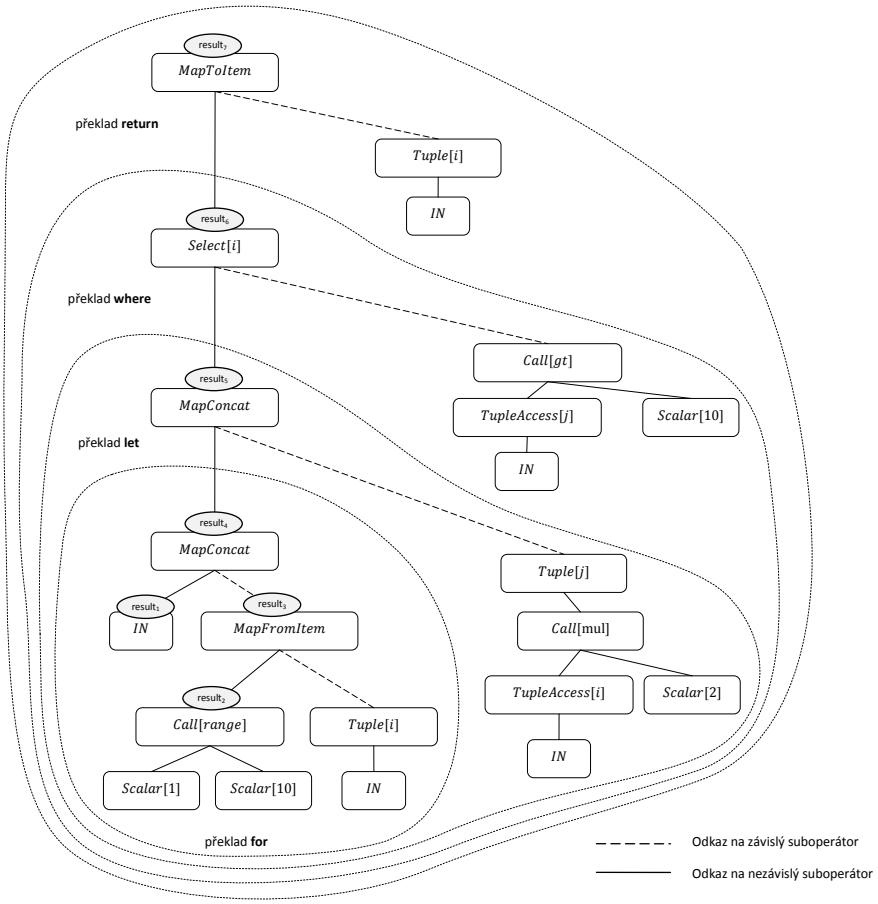
Během překladu bude postupně vytvářen strom na obrázku 7 níže. V tabulce 2 jsou uvedeny mezivýsledky při vykonávání dotazu, které se vztahují k vyznačeným místům ve stromu operátorů.

Překlad for a rozsahového výrazu

Překlad celého výrazu je zahájen kompilací `for`. Nejprve je přeložen výraz `1 to 10` na volání dvouparametrové funkce `range(low, high)`. Jedná se o jednoduché kompilační pravidlo pro překlad rozsahu hodnot, které zde dosud nebylo uvedeno. Funkce `range` vrací sekvenci celých čísel od definované spodní hodnoty podle parametru `low` po horní hodnotu `high` s krokem 1. Mezivýsledek po vyhodnocení Call[range] je možné vidět v tabulce 2 jako $result_2$. Operátor MapFromItem v kombinaci s Tuple[i] zajistí, že bude vzniklá sekvence transformována na tabulku s n-ticemi o jedné složce *i*. Výsledek po MapFromItem je označen jako $result_3$.

Celý dosavadní mezivýsledek byl počítán v rámci vyhodnocení závislé části operátoru MapConcat pro každou n-tici z jeho nezávislého vstupu, který je momentálně tvořen IN. Tento IN se nevztahuje k žádnému závislému vstupu jiného operátoru, ale k programové funkci, která

celé vyhodnocování zahájila (viz. kapitola 3.2.3). Pro zjednodušení můžeme předpokládat, že tato funkce nastavila kontextovou n-tici jako prázdnou ($result_1$). Výsledek operátoru MapConcat, resp. celé klauzule `for` je uveden v $result_4$.



Obrázek 7: Kompilace FLWOR

$result_1 = []$ $\quad$ $result_2 = (1,2,3,4,5,6,7,8,9,10)$

$result_3 =$

i
1
2
⋮
10

$result_4 =$

i
1
2
⋮
10

$result_5 =$

i	j
1	1
2	4
⋮	⋮
10	20

$result_6 =$

i	j
6	12
7	14
8	16
9	18
10	20

$result_7 = (6,7,8,9,10)$

Tabulka 2: Mezivýsledky vyhodnocování

Překlad let a aritmetického výrazu

Následuje překlad klauzule `let`. Nejprve je přeložen dílčí výraz `$i * 2`. Podle pravidla pro kompilaci aritmetického výrazu násobení v kapitole 3.3.1 se tento výraz překládá na volání funkce `mul`. První činitel je dán proměnnou *i*, místo operátoru pro přístup k proměnné Var[i] se díky substituci uplatní operátor TupleAccess[i] nad kontextovou n-ticí. Druhý činitel tvoří operátor generující skalární hodnotu Scalar[2]. Operátor MapConcat zajistí, že je tento připravený výpočet spuštěn nad každou n-ticí z jeho nezávislého vstupu, který je tvořen výstupním operátorem z překladu předchozí klauzule `for`. Mezivýsledek po vyhodnocení MapConcat, čili celé klauzule `let` ukazuje opět tabulka 2, proměnná $result_5$.

Překlad where s výrazem pro porovnávání

Porovnání > se podle odstavce 3.3.1 překládá na volání funkce `gt`. Nezávislý vstup selekce tvoří výstup předchozí klauzule `let`. Výsledek po selekci ukazuje v tabulce mezivýsledků proměnná $result_6$.

return

Poslední klauzulí je `return`. Nejprve je zkompilován výraz pro výpočet jedné dílčí položky. Jedná se pouze o přístup k proměnné, který se vzhledem k substituci převádí na přístup do příslušné složky kontextové n-tice. MapConcat tento výpočet spustí nad každou n-ticí z tabulky vzniklé předchozí klauzulí `where`. Všechny výsledné položky utvoří sekvenci, která je výstupem celého dotazu, viz. proměnná $result_7$.

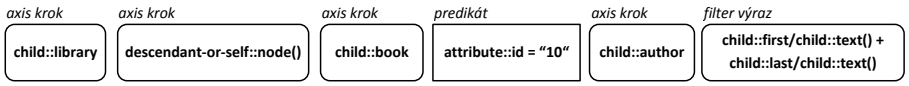
3.3.3 XPath výrazy

Pokud by algebra pracovala pouze s XQuery core, pak by se kompilace XPath výrazů omezila jen na zpracování axis kroků, jelikož predikáty a filter výrazy lze normalizací převést na FLWOR. Plán dotazu lze ale sestavit rovnou z XPath v původní podobě.

Gramatika povoluje používání zkratkových zápisů (viz. kapitola 2.3.1). Zkrácené zápisy lze ale jednoduše rozepsat na plný tvar, takže se jim dále věnovat nebudeme. Podrobnější informace lze nalézt v [4]. Na obrázku 8 je příklad XPath výrazu s použitím zkratkových zápisů a po rozepsání. Obrázek dále zachycuje rozklad výrazu na jednotlivé kroky a predikáty.

```
/library//book[@id = "10"]/author/(first/text() + last/text())

/child::library/descendant-or-self::node()/child::book[attribute::id = "10"]/child::author/
  (child::first/child::text() + child::last/child::text())
```



Obrázek 8: Struktura XPath výrazu

Stejně jako v případě FLWOR probíhá kompilace XPath po částech za použití kontextového operátoru.

Zahájení překladu

1. V případě, že se na začátku výrazu nachází rovnou axis step, je kontextový operátor inicializován na přístup k proměnné Var[dot].
2. Pokud je výraz uvozen jednoduchým lomítkem, začíná vyhodnocování od kořene XML dokumentu. Kořen můžeme snadno obdržet voláním vestavěné funkce Call[root]().
3. Jestliže se na začátku nachází dvojité lomítko, je potřeba načíst XML dokument a rovnou z něj vybrat všechny potomky pomocí TreeJoin[descendant-or-self, node()](Call[root]()).

Axis kroky

$$\frac{Op_{out} = \text{TreeJoin}[axis, nodetest](Op_0)}{[\![axis :: nodetest]\!]_{Op_0} \Rightarrow Op_{out}}$$

Kompilace axis step je přímočará. Stačí pouze aplikovat operátor TreeJoin, který je pro tento účel v podstatě určený. Operátor TreeJoin pracuje tak, že pro každý XML uzel ze vstupní sekvence provede požadovaný pohyb po dané ose (*axis*), čímž vznikne mezivýsledek obsahující množinu XML uzlů. Všechny mezivýsledky jsou nakumulovány do celkové výsledné (uspořádané) množiny, přičemž pořadí uzlů musí být takové, jako bylo v původním dokumentu a musí být eliminovány duplicitní výskyty. Eliminace duplicit je nutná z důvodu více následujících axis kroků, kdy by mohlo dojít k situaci, že se jeden konkrétní uzel vyskytne ve výsledné množině dvakrát.

Kromě toho je provedena filtrace výstupní množiny na základě daného testu uzlu (statický argument operátoru – *nodetest*). Jedná se nejčastěji o jmenný test elementu. Existuje ale celá řada testů – testy na atributy, textové uzly, komentáře, na libovolný uzel a další.

Činnost operátoru TreeJoin bývá nazývána jako *navigace*.

Filter výrazy

$$\frac{\begin{array}{c} filterExpr \Rightarrow Op_1 \\ Op_2 = \text{MapFromItem}\{\text{Tuple[dot]}(\text{IN})\}(Op_0) \\ Op_3 = \text{MapConcat}\{Op_2\}(\text{IN}) \\ Op_out = \text{MapToItem}\{Op_1\}(Op_3) \end{array}}{[\![filterExpr]\!]_{Op_0} \Rightarrow Op_{out}}$$

Nejprve je zkompilován samotný filter výraz na pomocný operátor Op_1. Poté je zajištěno převedení vstupní sekvence reprezentované operátorem Op_0 pomocí MapFromItem na tabulku, která je následně přimapována pomocí MapConcat k aktuální kontextové n-tici. Kontextová n-tice totiž může obsahovat nějaké platné proměnné např. vnějšího výrazu FLWOR. Nakonec je aplikován operátor MapToItem. Ten postupně nad n-ticemi v tabulce jednak provede samotný výpočet filter výrazu, který je v tuto chvíli zkompilovaný v Op_1, a jednak provede převedení zpět na sekvenci položek.

Predikáty

V případě kompilace predikátů algebra opět využije operátory pracující v prostoru n-tic. Úplně stejně jako u filter výrazů bude i zde potřeba zajistit dostupnost kontextového uzlu a všech aktuálně platných proměnných dvojicí operátorů MapFromItem a MapConcat.

Poté jednoduše vybereme vyhovující záznamy operátorem Select a provedeme přechod zpět od tabulky k sekvenci. Jedinou výraznější komplikací je, že se v predikátu nemusí nutně nacházet pouze logický výraz. Specifikace XPath totiž umožňuje, aby typ vnitřního výsledku predikátu byl kromě logického výrazu také celé číslo nebo sekvence.

Je-li vnitřním mezivýsledkem celé číslo, pak toto číslo představuje pořadí jediného uzlu ze vstupní sekvence, který bude propuštěn dále. Jinak rozhodne tzv. efektivní booleovská hodnota sekvence, pro jejíž výpočet platí následující pravidla:

1. Obsahuje-li sekvence jedinou položku typu *boolean*, pak efektivní booleovská hodnota je dána hodnotou této položky.
2. Obsahuje-li sekvence jedinou položku typu *integer* nebo *double*, nabývá efektivní booleovská hodnota pravdy právě když dané číslo není nulové.
3. Obsahuje-li sekvence jedinou položku typu *string*, je efektivní booleovská hodnota pravdivá v případě neprázdného řetězce.
4. V případě jakékoli obecné sekvence rozhodne o efektivní booleovské hodnotě její neprázdnost.

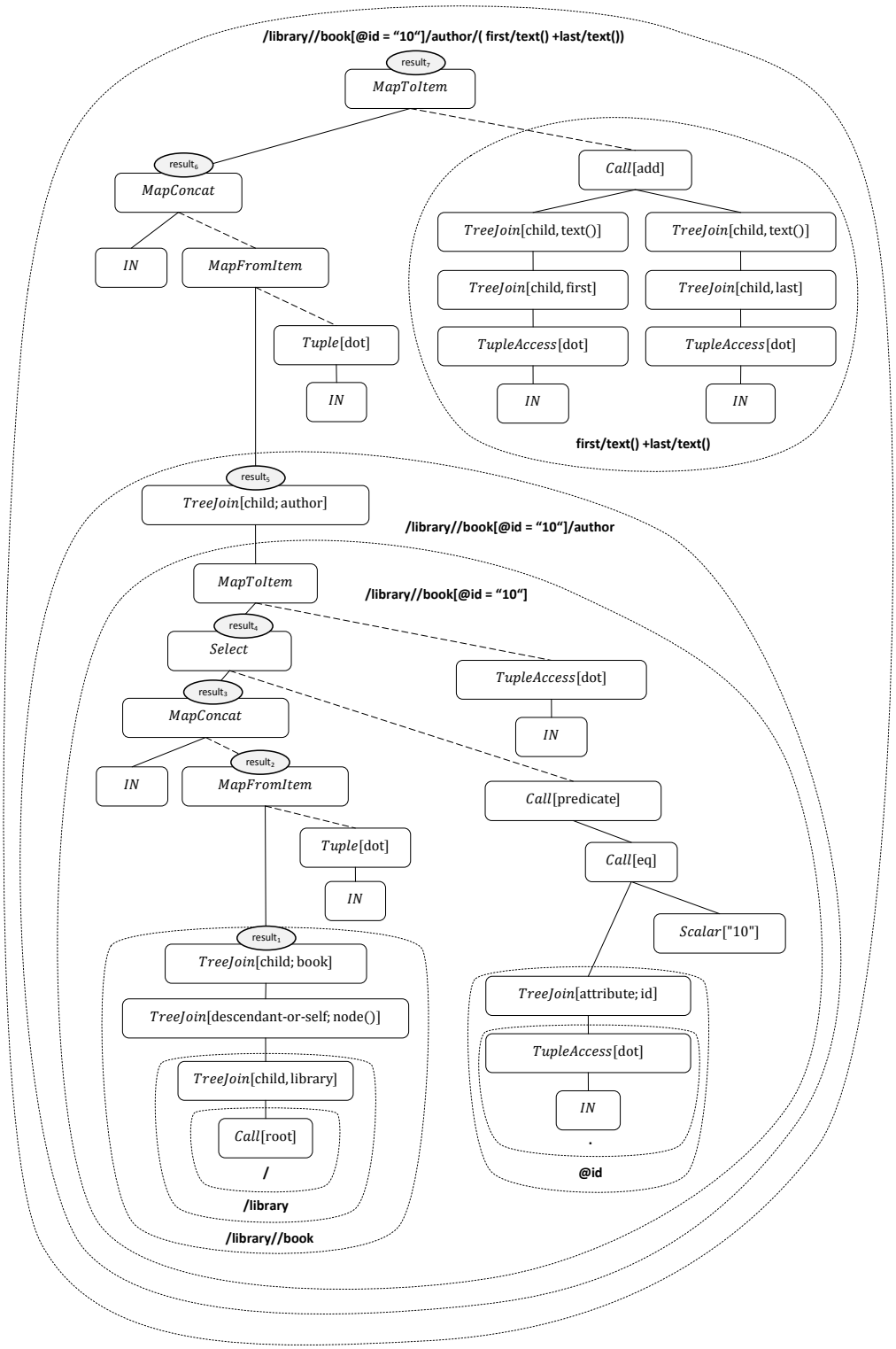
Normalizace převádí predikáty XPath na `where` klauzule FLWOR výrazů, přičemž pro alternativu dle datových typů využívá konstrukci `typeswitch`. Tenhle způsob je však relativně komplikovaný a přidává do celého výrazu zbytečně mnoho operátorů navíc. Pro celou logiku vyhodnocení pravdivostní hodnoty pro mezivýsledek predikátu byla vytvořena jednoduchá vestavěná funkce `predicate`.

$$\frac{\begin{array}{c} predicateExpr \Rightarrow Op_1 \\ Op_2 = \text{MapFromItem}\{\text{Tuple[dot]}(\text{IN})\}(Op_0) \\ Op_3 = \text{MapConcat}\{Op_2\}(\text{IN}) \\ Op_4 = \text{Select}\{\text{Call[predicate]}(Op_1)\}(Op_3) \\ Op_{out} = \text{MapToItem}\{\text{TupleAccess[dot]}(\text{IN})\}(Op_4) \end{array}}{[\![predicateExpr]\!]_{Op_0} \Rightarrow Op_{out}}$$

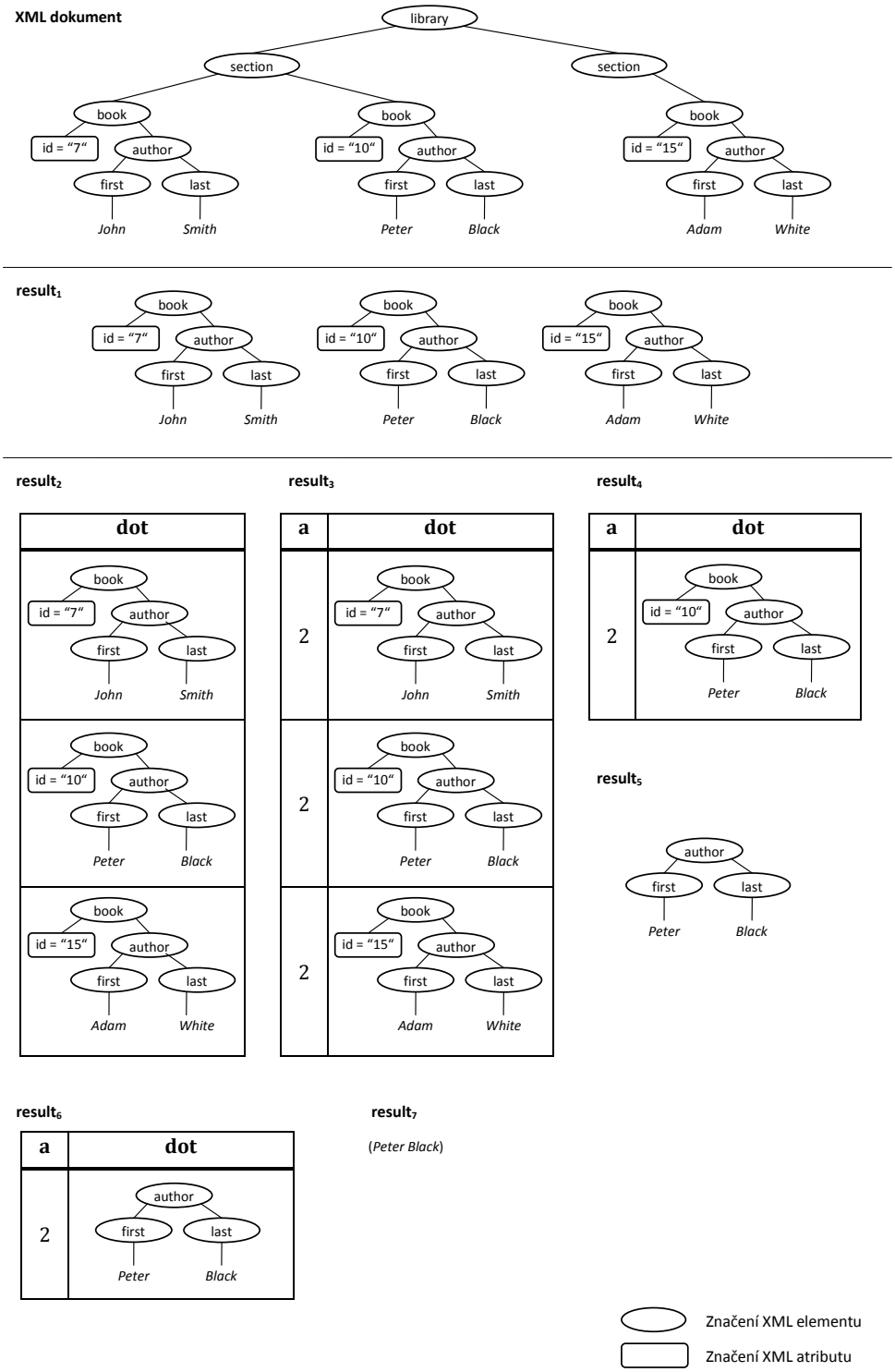
Kompilace tedy proběhne z hlediska převodu mezi n-ticemi a položkami stejně jako u filter výrazů, pouze dojde k omezení tabulky operátorem Select, kde jeho závislá část nevyhodnocuje operátor podmínky přímo, ale až na základě výsledku funkce `predicate`.

Příklad 3: *Překlad XPath výrazu*

Nadefinovaná kompilační pravidla budou prezentována na příkladu z obrázku 8 výše. Výsledný strom operátorů na obrázku 9 níže se vyznačenými místy odkazuje na obrázek 10, kde jsou schématicky znázorněny mezivýsledky vybraných důležitých operátorů.



Obrázek 9: Překlad ukázkového XPath výrazu



Obrázek 10: Mezivýsledky ukázkového překladu XPath

```
/child::library/descendant-or-self::node()/child::book[attribute::id = "10"]
  /child::author/(child::first/child::text() + child::last/child::text())
```

Výraz je uvozen jednoduchým lomítkem, takže je potřeba zajistit načtení vstupního XML dokumentu voláním funkce `root`. Prvním krokem je navigace na všechny přímé potomky s názvem „library“. Jedná se o axis step, takže použijeme operátor TreeJoin. Následuje zápis `/descendant-or-self:node()/child:book` (po rozepsání ze zkratkového `//book`). Jde tedy o dvojitý axis step, takže dvakrát použijeme TreeJoin. Při vyhodnocování v tuto chvíli obdržíme mezivýsledek označený jako $result_1$.

Další částí dotazu je predikát. Vznikající sekvenci položek je tedy nutné převést na tabulku s n-ticemi o jedné složce *dot*. O toto převedení se stará operátor MapFromItem s konstruktorem n-tice (operátorem Tuple) v závislé části (viz. mezivýsledek $result_2$). To vše proběhne v rámci závislé větve operátoru MapConcat, který zajistí, že v podmínce selekce budou dostupné kromě kontextového uzlu také všechny proměnné, které nastavil např. vnější výraz FLWOR. Můžeme předpokládat, že vnější kontextová n-tice obsahuje například složku *a* s hodnotou 2. Následuje provedení selekce ($result_4$) a návrat k toku datových položek operátorem MapToItem. Dalším krokem je již několikrát překládaný axis step se jmenným testem na název uzlu „author“ a navigaci na osu *child*, takže opět jednoduše použijeme operátor TreeJoin.

Posledním krokem je filter výraz. Zde je možné opět pozorovat přechod od sekvence položek k tabulce pomocí operátorů MapFromItem, Tuple a MapConcat na stejném principu jako při kompilaci predikátu. Pro každou n-tici pak proběhne výpočet filter výrazu, o což se postará závislá větev operátoru MapToItem. Výsledek dotazu je označený jako $result_7$. Je zde pro zjednodušení drobná nepřesnost. Při spojování křestního jména a příjmení by se mezi oběma částmi samozřejmě měla nacházet mezera.

3.3.4 Kompilace dalších konstrukcí

Byl popsán překlad nejpoužívanějších konstrukcí XQuery. Mezi další méně známé konstrukce paří např. již zmiňovaný `typeswitch`, který provádí větvení na základě datového typu. Dále pak klasická alternativa, tzn. větvení na základě booleovské hodnoty, všeobecné kvantifikátory `every`, existenční kvantifikátory `some` a další. Kompilační pravidla jsou pro tyto konstrukce většinou přímočará a využívají operátory pracující na n-ticemi. Jejich podrobný popis by byl pro účely této práce zbytečně zdlouhavý.

3.4 Optimalizace

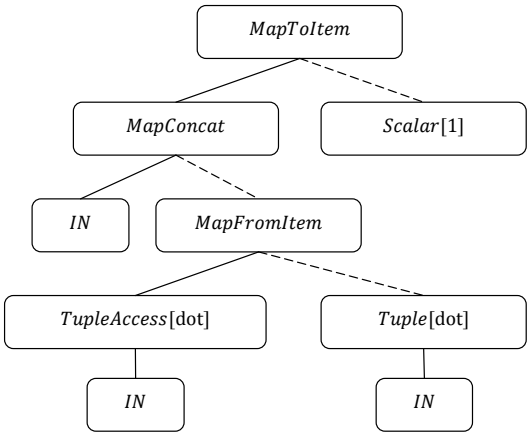
Výše popsaná pravidla pro kompilaci sice sestaví strom operátorů tak, abychom po jeho evaluaci obdrželi správný výsledek, nicméně ne vždy je výpočet efektivní. Existuje celá řada pravidel, jak operátory přeskládat, aby výsledek zůstal zachován, ale výpočet proběhl rychleji. Bude popsáno celkem 5 pravidel, z nichž první dvě jsou specifická pro tuto práci.

3.4.1 **Odstranění** MapConcat

Tato optimalizace se týká především částí operátorového stromu, které vznikly kompilací XPath výrazů. Problémem je, že každý byť elementární výraz, např. konstanta, představuje z hlediska gramatiky jednoduchý XPath s jedním filter výrazem. Z toho vyplývá nutnost namapovat výsledek z předchozího nebo počátečního kroku na proměnnou *dot*.

Na obrázku 11 je zachycen strom operátorů pro výraz „`1`“. Jedná se o prakticky nepoužitelný dotaz, nicméně z hlediska gramatiky XQuery je vše naprosto v pořádku. Už na první pohled je zřejmé, že kompilační pravidla byla v tuto chvíli zbytečně důsledná a vygenerovala velké množství přebytečných operátorů.

První optimalizační pravidlo, které by nás v tuto chvíli mohlo napadnout je odstranit celý MapToItem a zanechat pouze Scalar[1], jelikož skalární operátor nepřistupuje ke kontextové n-tici, tzn. nezávislý vstup vůbec nepotřebuje. Tato úvaha ale není v pořádku, protože kdyby nezávislá část vyprodukovala více než jednu n-tici, pak by výsledek nebyl správný. Závislá část se musí provést přesně tolikrát, jaký je počet n-tic z nezávislého operátoru.



Obrázek 11: Kompilace skalární hodnoty bez optimalizace

Zkusme se nejprve zaměřit na činnost suboperátoru MapFromItem. V nezávislém suboperátoru pouze přistoupíme k nějaké složce *dot* kontextové n-tice, obdržíme tedy položku, kterou předáme závislému vstupu, který z ní vytvoří novou jednoprvkovou n-tici. Situace je ilustrována na tabulce 3. Nalevo je obsah kontextové n-tice, ke které přistupoval TupleAccess, napravo výsledek MapFromItem.

a	**b**	**dot**	⟶	**dot**
1	2	`<item />`		`<item />`

Tabulka 3: Mezivýsledek MapFromItem

N-tici, ke které přistupoval TupleAccess nastavil jako kontextovou operátor MapConcat. Její obsah musí být shodný s obsahem nějaké vnější kontextové n-tice, která shodou okolností utvořila vstup tohoto operátoru. Výsledek MapConcat je uveden v tabulce 4.

a	b	dot	dot
1	2	<item />	<item />

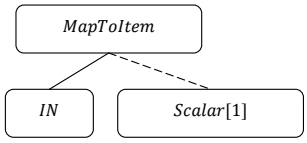
Tabulka 4: Mezivýsledek MapConcat

Názvy složek n-tic se sice dle definice nesmí opakovat, nicméně procesor si s duplicitními výskyty poradí tak, že přistupuje vždy k poslednímu z atributů v pořadí přidávání. V tuto chvíli je ale jasné, že došlo k pouhému zkopírování jedné složky a to je z hlediska dalšího vyhodnocování zbytečné. Pokud by některý z operátorů později potřeboval přistoupit k *dot*, nebude záležet na tom, kterou kopii skutečně použije.

Optimalizační pravidlo tedy funguje tak, že je li nalezen MapConcat s nezávislým vstupem IN a závislým vstupem tvořeným MapFromItem, kde dochází pouze k přístupu ke složce q a konstrukci jednoprvkové n-tice se složkou q, může být tento MapConcat rovnou nahrazen operátorem n-ticí IN.

$$\text{MapConcat}\{\text{MapFromItem}\{\text{Tuple}[q](\text{IN})\}\}(\text{TupleAccess}[q](\text{IN}))\}(\text{IN}) \rightarrow \text{IN}$$

Po aplikaci tohoto optimalizačního pravidla na strom operátoru z obr. 11, obdržíme nový strom na obr. 12.



Obrázek 12: Kompilace skalární hodnoty po optimalizaci odstraněním MapConcat

3.4.2 **Odstranění** MapToItem

Vraťme se nyní k původní úvaze odstranit MapToItem, pokud jeho závislá část nepoužívá kontextovou n-tici IN. Bylo řečeno, pokud by vyhodnocením nezávislého suboperátoru vznikla tabulka s jiným počtem n-tic než 1, nebyl by výsledek po odstranění MapToItem korektní, jelikož výsledek není ovlivněn jen obsahem, ale také délkou tabulky.

Jenže v tuto chvíli délku tabulky známe – jedná se přesně o jednu n-tici (kontextovou), takže výsledek odstranění operátoru MapToItem neutrpí žádnou újmu. Při dalším zamyšlení je možné MapToItem odstranit i v případě, že závislý vstup IN využívá. Za předpokladu, že nezávislým vstupem je IN, provádí MapToItem pouze zkopírování této kontextové n-tice opět do svého závislého vstupu.

$$\text{MapToItem}\{Op_1\}(\text{IN}) \rightarrow Op_1$$

Pokud tedy na strom z obrázku 12 uplatníme toto optimalizační pravidlo, zůstane pouze operátor Scalar[1]. Závěrečná část této práce (kap. 5.2) ukazuje časová srovnání pro běh dotazu bez a s optimalizacemi odstraněním MapConcat a MapToItem.

3.4.3 **Vložení** Product

Principem je nahrazení operátoru MapConcat kartézským součinem Product. Činnost těchto dvou operátorů se liší pouze tím, že MapConcat provádí připojení pravé tabulky ke každému záznamu levé tabulky, přičemž obsah pravé závisí na obsahu záznamů z levé. U operátoru Product jsou obě tabulky nezávislé.

$$\text{MapConcat}\{Op_2\}(Op_1) \rightarrow \text{Product}(Op_1, Op_2), \text{ za předpokladu, že } Op_2 \text{ je nezávislý na kontextu.}$$

Pokud tedy v závislém suboperátoru Op_2 nepoužijeme přístup k mezivýsledku nezávislého Op_1, můžeme nahradit MapConcat operátorem Product.

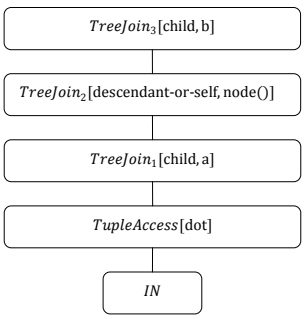
3.4.4 **Vložení** Join

Předchozí optimalizace tvoří v podstatě pouze mezikrok, který umožní použití operátoru Join dobře známého z relační algebry. Pokud se nad kartézským součinem Product nachází operátor Select, je možné tuto dvojici nahradit operátorem Join. Výhoda tohoto nahrazení spočívá v tom, že lze v závislosti na tvaru spojovací podmínky pro evaluaci Join použít různé efektivní algoritmy, např. spojení hashováním (hash-join). Optimalizace vložením Product a Join jsou převzaty z [16].

$$\text{Select}\{Op_1\}(\text{Product}(Op_2, Op_3)) \rightarrow \text{Join}\{Op_1\}(Op_2, Op_3)$$

3.4.5 Spojení kroků XPath

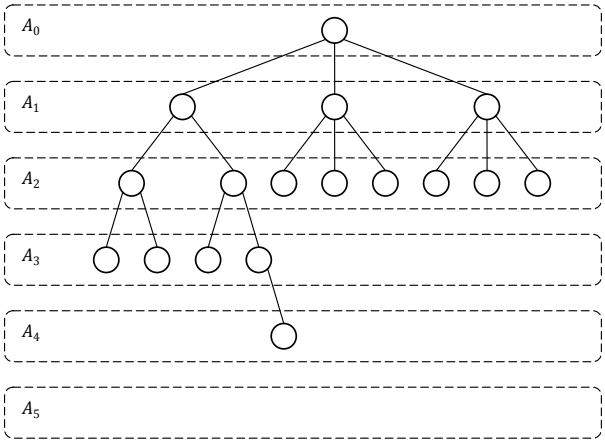
Poslední zde popsaná a později implementovaná optimalizace souvisí s používáním zkratkového zápisu osy *descendant-or-self* pomocí `//`. Jednoduchý XPath `a//b` je rozepisován na `a/descendant-or-self::node()/child::b`. Na obrázku 13 je sestavený plán dotazu pro vyhodnocení takového výrazu, pro snazší orientaci jsou jednotlivé operátory TreeJoin rozlišeny spodním indexem.



Obrázek 13: Plán dotazu `a//b`

Optimalizační pravidlo jednoduše transformuje dva po sobě jdoucí operátory TreeJoin – první s osou *descendant-or-self* a testem na libovolný XML uzel (mimo atribut), druhý s osou *child* a testem q, na jeden TreeJoin s osou *descendant* a testem q.

TreeJoin[descendant-or-self, node()](TreeJoin[child, q](Op)) → TreeJoin[descendant, q](Op)



Obrázek 14: Rozbor optimalizace

Důkaz

Na obrázku 14 je znázorněn podstrom nějaké části XML dokumentu. Pro odvození tohoto přepisu bude potřeba formálně nadefinovat několik pomocných funkcí. Funkce c(a) přiřazuje každému uzlu a množinu jeho přímých potomků A. Funkce child(A) předchozí zobrazení pouze zobecňuje pro možnost použití nad celou množinou podle vztahu (1). Jedná se tedy o popis činnosti operátoru TreeJoin[child, node()](A)

$$\text{child}(A) = \bigcup_{a \in A} \text{c}(a) \tag{1}$$

Pohledem na obrázek 14 by mělo být jasné, že výsledkem zobrazení child pro každou úroveň je úroveň následující (2).

$$A_{i+1} = \text{child}(A_i) \tag{2}$$

Dále je potřeba formálně zaznačit činnost TreeJoin při navigaci na osy *descendant-or-self* a *descendant* (funkce dos(A) a desc(A)). Úroveň 0 bude považována jako referenční, ke které budeme jednotlivé navigace vztahovat.

$$\text{dos}(A_0) = \bigcup_{i=0}^{n} A_i \tag{3}$$

$$\text{desc}(A_0) = \bigcup_{i=1}^{n} A_i \tag{4}$$

Každý XML dokument je samozřejmě konečný, což znamená, že musí existovat nějaká úroveň n, pro kterou platí (**??**).

$$A_n = \emptyset \tag{5}$$

Je tedy potřeba dokázat (**??**), tzn., že provedení navigace na osu *descendant-or-self* následované navigací na *child* je ekvivalentní k provedení jedné navigace *descendant*.

$$\text{child}(\text{dos}(A_0)) = \text{desc}(A_0) \tag{6}$$

Na levou i pravou stranu vztahu (3) aplikujeme navigaci *child* a obdržíme vztah (7).

$$\text{child}\left(\text{dos}(A_0)\right) = \text{child}\left(\bigcup_{i=0}^{n} A_i\right) \tag{7}$$

Následuje klíčový bod, kdy se s funkcí pro navigaci na *child* zanoříme do množinového sjednocení. Nezáleží na tom, zda navigaci na *child* provedeme nad celou množinou po sjednocení jednotlivých úrovní nebo provedeme navigaci nad každou úrovní zvlášť a sjednocení provedeme až pak.

$$\text{child}\left(\text{dos}(A_0)\right) = \bigcup_{i=0}^{n} \text{child}\left(A_i\right) \tag{8}$$

Za použití vztahu (2) provedeme jednoduchou úpravu na tvar (7) a (8).

$$\text{child}\left(\text{dos}(A_0)\right) = \bigcup_{i=0}^{n} A_{i+1} \tag{9}$$

$$\text{child}\left(\text{dos}(A_0)\right) = \bigcup_{i=1}^{n+1} A_i \tag{10}$$

Horní hranici pro množinové sjednocení $n + 1$ je možné snížit na n, jelikož už množina A_n je prázdná, takže A_{n+1} musí být prázdná také. Potom pravá strana odpovídá podle vztahu (4) navigaci na osu *descendant* a je tedy odvozen vztah (**??**).

V závěru této práce jsou opět v kapitole 5.2 srovnány časy vyhodnocování dotazu bez a s touto optimalizací.

3.4.6 Další možnosti optimalizace

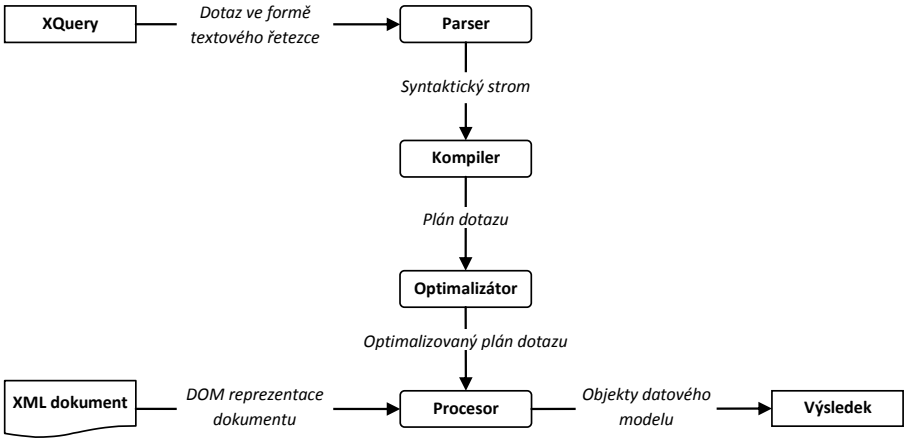
Další možnosti optimalizace, které v této práci nejsou implementovány, spočívají v přesunu některých operátorů v plánu dotazu směrem ke kořenovému operátoru nebo směrem k listům. V článku [16] je popsán princip tzv. *unnesting* přepisů, jejichž účelem je pokud možno eliminovat vykonávání zanořených smyček při vyhodnocování dotazu, což se týká především činnosti operátorů MapConcat. Toho je docíleno zavedením speciálního operátoru GroupBy a několika optimalizačních pravidel, které např. umožní přesun GroupBy ze závislého suboperátoru MapConcat nad tento MapConcat.

Efektivnějšího vyhodnocení by bylo možné dosáhnout také zavedením operátoru TupleTreePattern [14], který je schopen v jednom kroku efektivně vyhodnotit více za sebou

následujících axis steps a predikátů např. za použití indexované XML databáze. Operátor TupleTreePattern pracuje na základě tzv. TPQ (Tree Pattern Query), což je zjednodušená podoba XPath výrazů. Princip optimalizace tedy spočívá v nalezení co největších možných částí v plánu dotazu, které lze přepsat na TupleTreePattern, jemuž je statickým atributem nadefinován TPQ.

4 Implementace XQuery procesoru

Tato kapitola se věnuje postupu implementace XQuery procesoru, který je součástí přílohy diplomové práce. Zjednodušená bloková funkce celého procesoru je zachycena na následujícím schématu.



Obrázek 15: Bloková struktura procesoru

Na vstupu vždy stojí zadaný XQuery dotaz, který je v prvním kroku převeden parserem na syntaktický strom. Ze syntaktického stromu dále kompiler sestaví plán vykonávání dotazu, který je případně poupraven optimalizátorem. V poslední fázi vstupuje do hry procesor, který strom operátorů vyhodnotí, přičemž může, ale také nemusí, využívat kořenový XML dokument. Kapitola se bude postupně zaměřovat na detailnější jednotlivých bloků z hlediska implementace.

4.1 Standard W3C a zjednodušení

Tato implementace standard jazyka XQuery z velké části dodržuje, nicméně objevuje se zde několik úprav a zjednodušení.

4.1.1 Podporované konstrukce

1. FLWOR výrazy

   FLWOR výrazy jsou podporovány v celém rozsahu. Jediným omezením je chybějící možnost stanovení datového typu proměnné pomocí klíčového slova `as`. Toto omezení vychází ze zjednodušeného typového systému, který bude popsán později v kapitole 4.4.

2. XPath výrazy

   XPath výrazy jsou podporovány včetně všech definovaných XML os a zkratkových zápisů. Jsou podporovány jmenné testy a v omezené míře také tzv.

kind tests (tj. testy na typ uzlu), konkrétně `comment()`, `node()`, `text()` a `attribute(attribName)`

3. Aritmetické a relační výrazy

   Jsou podporovány standardní aritmetické výrazy nad číselnými položkami – operace sčítání, odčítání, násobení, dělení (celočíselné i desetinné) a modulo. Dále standardní výrazy porovnávání – ostré a neostré uspořádání, rovnost a nerovnost položek. Vzhledem k tomu, že implementace není omezena pouze na XQuery core, při používání není nutné přepisovat aritmetické nebo relační operátory na vestavěné funkce. Tato implementace je oproti standardu mírně náročnější na dodržování typových konvencí. Zejména při práci s aritmetickými výrazy je potřeba provádět explicitní přetypování hodnot zadaných textovými řetězci nebo XML uzly na celé nebo desetinné číslo.

4. Logické výrazy

   Podporovány jsou základní logické operace – logický součet `or` a součin `and`. Operace negace je v XQuery standardně realizována vestavěnou funkcí `not`.

5. Alternativa

   Implementace podporuje větvení `if` a větvení podle datového typu `typeswitch`.

6. Přetypování

   V rámci vydefinovaného zjednodušeného typového systému, je podporována i operace přetypování. Kompletní popis kompatibility a přetypovatelnosti jednotlivých typů je možné nalézt v příloze B.

7. Kvantifikované výrazy

   Podporovány jsou také všeobecné a existenční kvantifikované výrazy `every` a `some`.

8. XML konstruktory

   Implementace podporuje konstruktory XML uzlů a to jak konstruktory přímé, tak konstruktory vypočtené (viz. kapitola 2.3.2).

9. Vestavěné funkce

   Kompletní seznam podporovaných vestavěných funkcí je uveden v tabulce 5. Jejich popis včetně typů argumentů lze nalézt v příloze A. Význam funkcí neodpovídá přesně standardu a je upraven pro potřeby této práce. Je velmi pravděpodobné, že bude tento seznam v budoucnu rozšiřován.

add	sub	mul	div	idiv	mod	eq	ne
lt	gt	le	ge	veq	vne	vlt	vgt
vle	vge	abs	count	position	last	doc	root
predicate	boolean	range	distinct	avg	sum	min	max
empty	not						

Tabulka 5: Podporované vestavěné funkce

4.1.2 Nepodporované nebo částečně podporované konstrukce

- Vytváření funkcí a modulů
- Specifikace uživatelem definovaných typů
- Jmenné prostory
- Ordered a unordered výrazy
- Procesní instrukce, entity a sekce CDATA
- Porovnávání pořadí XML elementů

4.2 Implementační prostředí

Celý XQuery procesor byl naimplementován v jazyce C++. Bylo využito vývojové prostředí Microsoft Visual Studio (v průběhu vývoje ve verzích 2010 a 2005). Jazyk C++ byl zvolen zejména z důvodu rychlosti běhu výsledného programu a možnosti efektivního nakládání s pamětí. Implementace by pravděpodobně byla pohodlnější v některém z vyšších programovacích jazyků - C# nebo Java, avšak za cenu ztráty zde velmi žádané rychlosti.

Kromě základních datových typů a funkcí, které C++ nabízí, byl využíván jmenný prostor `std` především pro práci s textovými řetězci. Později byla využita i externí knihovna Xerces Parser pro načítání vstupních XML dokumentů.

4.3 Univerzální datové struktury

Na začátku i během vývoje procesoru vzniklo několik univerzálních datových typů především pro zjednodušení práce s polem. Později se bude text na tyto vzniklé typy často odvolávat, proto tedy alespoň stručný popis.

4.3.1 MyList

Jedná se o šablonu zapouzdřující práci s dynamickým polem pomocí metod jako `add(T item)`, `removeAt(int index)`, `T getItem(int index)` a `setItem(int index, T item)`. Při přidávání prvků se `MyList` v případě nutnosti sám stará o správné přealokování interního pole. Z hlediska efektivity přealokování neprobíhá vždy při přidávání prvku. Pokud současná kapacita nestačí, dojde k alokaci nového pole o dvojnásobném rozměru. Experimentálně se ukázalo, že nejvhodnější výchozí velikost je 1. Při vytváření instance však lze počáteční kapacitu určit argumentem konstruktoru.

4.3.2 MyStack

Význam této šablony je evidentní z názvu. Jedná se o zásobník implementovaný dvěma způsoby – spojovým seznamem a polem. Konkrétní implementaci je možno zvolit při kompilaci zdrojových kódů procesoru. Jsou podporovány standardní metody `push(T item)`, `T pop()`, `T getPeak()` a `bool empty()`. Jejich význam ne třeba vysvětlovat. Pro realizaci seznamu bylo nutné vytvořit třídu `MyLinkNode` reprezentující jeden uzel jednosměrného spojového zenamu, který zapouzdřuje skutečně nesenou hodnotu a přidává k ní odkaz na následující prvek.

4.3.3 MyQueue

Opět velmi dobře známá datová struktura – fronta s podporou metod `enqueue(T item)`, `T dequeue()`,`T getHead()`,`T getTail()` a `bool empty()`. Implementaci vnitřně zajišťuje spojový seznam pomocí třídy `MyLinkNode` popsané v předchozím odstavci.

4.3.4 MyHashSet

Jedná se o implementaci množiny s metodami `add(T item)` a `bool contains(T item)`. Z názvu vyplývá, že `MyHashSet` pro realizaci množiny vnitřně využívá hash tabulku. Pro řešení kolizí je zde využívána metoda dvojitého hashování s funkcemi $h_1(k) = k \bmod m$ a $h_2(k) = 1 + (k \bmod (m-1))$, kde k představuje celočíselnou hodnotu identifikující prvek a m velikost tabulky.

Tabulka má ve výchozím stavu 11 volných pozic. K přealokování dojde vždy po dosažení poloviční zaplnění. V rámci třídy je k dispozici seznam 30-ti předdefinovaných prvočísel, podle kterých se v případě potřeby určí nová velikost. Prvočísla jsou po sobě zvolena tak, aby každé bylo přibližně 2 krát větší, viz. tabulka 6.

11	23	47	97	197
397	797	1597	3203	6421
12853	25717	51437	102877	205759
411527	823117	1646237	3292489	6584983
13169977	26339969	52679969	105359939	210719881
421439783	842879579	1685759167	3371518343	2448069391

Tabulka 6: Prvočísla pro volbu velikosti tabulky hash

Identifikaci prvku v množině zajišťuje pomocná šablona `MyEqComparer`, která abstraktně definuje metody pro detekci shodnosti a zjištění hodnoty hash. Vzhledem k tomu, že se prvky obvykle identifikují svou adresou, je součástí univerzálních struktur také implementace `ReferenceEqComparer`.

`MyHashSet` bylo nutné naimplementovat primárně pro efektivní práci operátoru *TreeJoin*, který v posledním kroku vyhodnocování eliminuje duplicity.

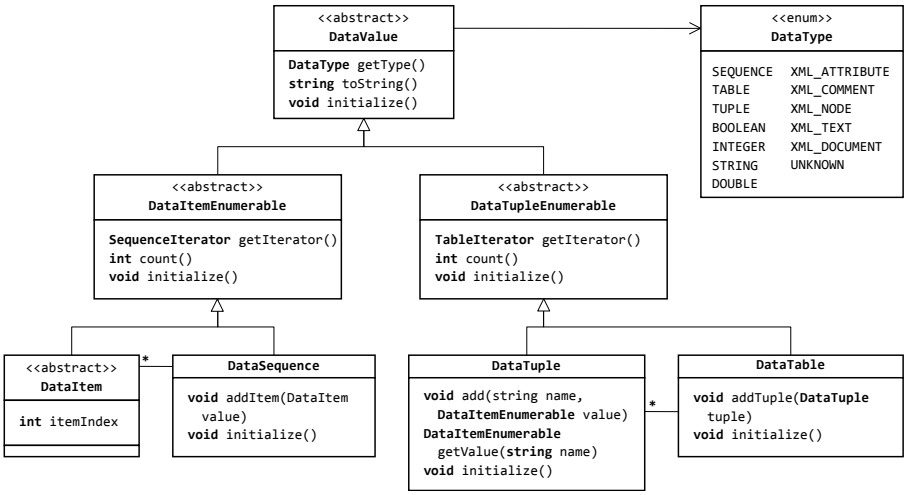
4.3.5 MyIterator

Jedná se o abstraktní třídu, která zajišťuje průchod polem, spojovým seznamem nebo jiným typem představujícím seznam hodnot pomocí metod `reset()`, `T getNext()` a `bool hasNext()`.

Tato abstraktní třída byla implementována pro `MyList` jako `MyListIterator` a pro `MyQueue` jako `MyQueueIterator`. Iterátory umožnily především v operátorech XQuery algebry oddělit procházení seznamu (např. sekvence položek nebo tabulky) od jeho fyzické implementace.

4.4 Datový model

Datový model popisuje systém navzájem propojených tříd, které zapouzdřují práci s jednotlivými typy hodnot XQuery.



Obrázek 16: Třídní diagram datového modelu

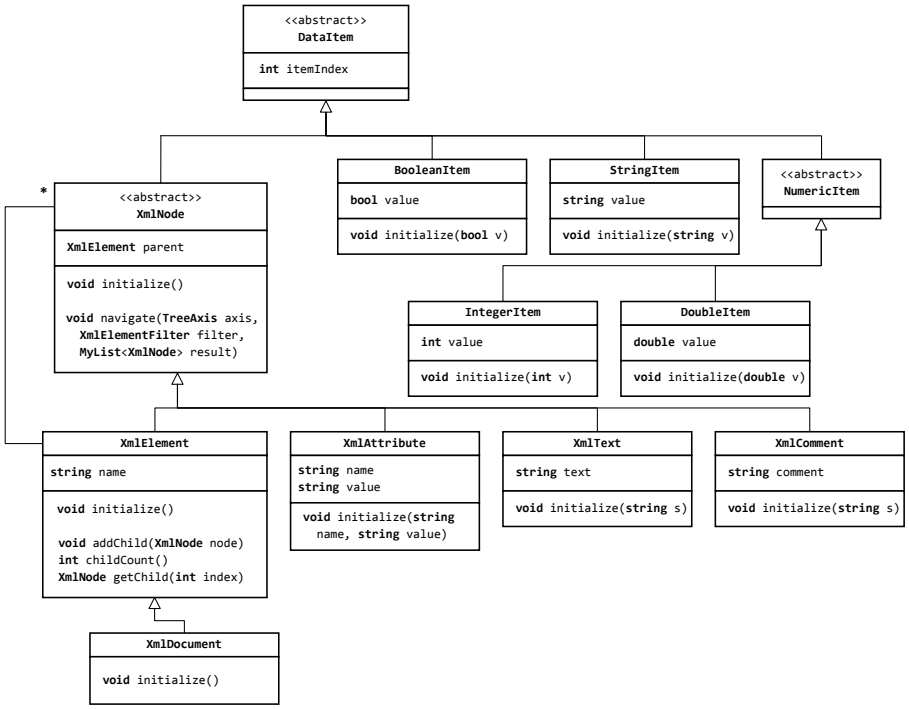
Z třídního diagramu na obrázku 16 je patrné, že jakákoli hodnota může být na nejvyšší úrovni abstrakce interpretována jako `DataValue`. Další úroveň dědičnosti spočívá v rozdělení na `DataTupleEnumerable` a `DataItemEnumerable`. V základu jde o duální struktury – jedna pro práci s n-ticemi, druhá pro práci s položkami.

Význam tříd `*Enumerable` spočívá v možnosti iterování pomocí `TableIterator` nebo `SequenceIterator`. Dochází tak k zakrytí rozdílů, zda pracujeme s n-ticí `DataTuple` nebo tabulkou `DataTable` a na druhé straně s položkou `DataItem` nebo sekvencí `DataSequence`. Třídní diagram datového modelu počínaje `DataItem` pokračuje na obrázku 17.

Uvedené třídy na obrázcích 16 a 17 slouží jako pevný základ, u kterého nejsou předpokládány pravidelné změny nebo rozšiřování. Z toho vychází také filosofie názvosloví a organizace některých metod.

4.4.1 Typový systém

Z pohledu dotazu, který procesor vykonává je pro každý datový objekt důležitý jeho typ daný výčtem `DataType`. Zde dochází ke zjednodušení oproti standardu W3C, který definuje možnost tvorby uživatelských typů. V této implementaci dotazy pracují pouze s fixní množinou datových typů, jejichž kompatibilita je popsána v příloze B. Z tohoto zjednodušení vyplývají některá další



Obrázek 17: Třídní diagram datové položky

omezení, např. nepodporovaná klauzule `as` ve výrazech FLWOR, kde uživatel sám specifikuje typ sekvence položek.

Jazyk XQuery je relativně silně typovaný, což znamená, že operace jako sčítání čísla s řetězcem reprezentujícím číslo jsou obvykle neplatné a je nutné využít nějakého mechanismu přetypování. Bylo nutné vyřešit konverzi mezi typy jak na implementační úrovni, tak na úrovni procesoru z uživatelského hlediska.

Přetypování v rámci implementace

Vzhledem k tomu, že při používání datového modelu dochází k přetypování velmi často, byl naimplementován relativně jednoduchý systém pomocných metod `is*` a `as*`. Pro každý typ modelu (včetně těch abstraktních) je ve třídě DataValue abstraktně definována metoda `as*` (např. `asDataItem()`), která provede přetypování bez nutnosti použití standardní závorkové syntaxe. Pro otestování, zda je přetypování vůbec možné slouží odpovídající abstraktní metoda `is*` (tzn. např. `isDataItem()`).

Význam ovšem není pouze v usnadnění a zpřehlednění zápisu. Uvedené metody totiž neprovádí přetypování vždy pouze ve smyslu klasického polymorfismu, ale například pro `StringItem` je možné zavolat `asDataSequence()`, přestože se `DataSequence` a `StringItem` nachází v hierarchii dědičnosti na jiné větvi. Návratová hodnota `is*` navíc nemusí být pro dva dané typy vždy stejná. Pokud např. zavoláme `isDataItem()` na `DataSequence`, můžeme obdržet hodnotu „pravda“ v závislosti na tom, zda sekvence obsahuje jedinou položku.

Přetypování z hlediska procesoru

Na druhé straně je nutné řešit přetypování na úrovni XQuery procesoru. Všechny třídy modelu implementují metody `implicitlyCastable(DataType type)`, a `castAs(DataType type)`, které tvoří základ operátorů Cast a Castable.

Procesor tedy podporuje konstrukce `castable` a `cast as`, přičemž je možné používat typy uvedené v tab. 7. Ve srovnání s výčtem `DataType` (třídní diagram na obr. 16) zde chybí typ pro tabulku a n-tici. Důvodem je, že tyto dva typy hodnot slouží výhradně pro mezivýsledky procesoru a nejsou tak přístupné uživateli, který spouští dotaz. Výčet navíc obsahuje pomocnou konstantu `UNKNOWN`.

`sequence`	Sekvence položek
`boolean`	Hodnota „true“ / „false“
`integer`	Celé číslo
`string`	Textový řetězec
`double`	Desetinné číslo
`xml-attribute`	Xml atribut
`xml-comment`	Xml komentář
`xml-node`	Xml element
`xml-text`	Xml textový uzel
`xml-document`	Xml dokument

Tabulka 7: Datové typy procesoru

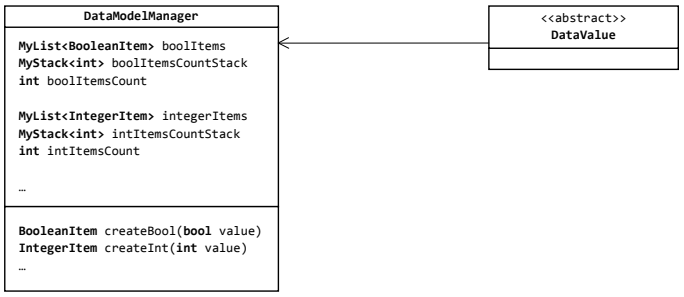
4.4.2 Správa datových objektů

O vytváření instancí datových objektů se stará třída `DataModelManager`, která pro každý (neabstraktní) datový typ obsahuje metodu `create*` (viz. třídní diagram na obr. 18). Každá nová hodnota si navíc zapamatuje odkaz na `DataModelManager`, který ji vytvořil, aby mohla případně sama zakládat další hodnoty pro výsledky výpočtů např. aritmetických operací. V úvahu by mohlo připadat také singleton řešení, ale z důvodu např. testování se může hodit vytvořit dva oddělené správce.

Samotné konstruktory datových typů zde nemají žádné argumenty, o veškerou inicializaci se starají metody `initialize(`*args*`)`. Inicializace jedné instance totiž může proběhnout vícekrát. Argumenty této metody se liší u finálních tříd, např. pro `BooleanItem` je inicializační metodě předávána skutečná hodnota typu `bool`, pro `XmlAttribute` dvě hodnoty typu `string` (`name` a `key`) atd. V rámci hierarchie dědičnosti je zajištěno, že při volání `initialize` na nějaký konkrétní finální typ budou zavolány také inicializační metody všech předků podobně jako se tomu děje při volání konstruktoru.

Recyklace instancí

Význam `DataModelManager` nespočívá pouze v evidenci referencí na vzniklé objekty, aby je nakonec bylo možné řádně zlikvidovat. `DataModelManager` umí existující objekty za určitých okolností znova využít.



Obrázek 18: Třídní diagram správy datových objektů

Ke všem neabstraktním typům datového modelu je přidruženo jednak zvláštní pole, resp. pole zapouzdřené třídou `MyList`, a jednak index poslední platné instance. Při požadavku na novou instanci dojde k inkrementaci indexu a zjištění, zda nebyla překročena velikost příslušného pole. Pokud ano, pak skutečně dojde k vytvoření a vložení do pole, jinak se využije již existující objekt, který je na daném indexu zrovna k dispozici.

Podle popisu v předchozím odstavci by ale proměnná ukazovala pokaždé na poslední pozici a nová instance by byla vytvářena vždy. To sice platí, ale pouze do té doby, než je na `DataModelManager` zavolána metoda `clear`. Ta jednoduše všechny indexy vynuluje a vzniklé objekty je možné začít využívat znova.

Tento způsob má ale relativně velkou nevýhodu. Zavoláním `clear` označíme jako neplatné všechny vzniklé datové objekty a tím pádem jediným bodem, kde je možné recyklaci využít, je zpracování více XQuery dotazů za sebou. Vyčištěním navíc přijdeme např. také o hodnoty, které vznikly zpracováním vstupního XML, takže před každým dotazem je nutné vstupní XML načítat znova.

Předchozí myšlenka byla upravena tak, že je s každým počítadlem asociován navíc ještě jednoduchý zásobník (implementovaný třídou `MyStack`) a metoda `clear` je nahrazena dvojicí metod `pushInstanceCounters` a `popInstanceCounters`. Pomocí zásobníku tak nedochází k úplnému vynulování, ale pouze k obnovení nějakého předchozího stavu.

Nejen, že je tímto způsobem odpadá nutnost znovu načítat vstupní XML, recyklaci lze navíc relativně snadno využít i přímo během vykonávání dotazů, což bude podrobněji ukázáno na vyhodnocování operátoru *Select* v kapitole 4.8.2.

4.4.3 Implementace DataSequence a DataTable

U implementace `DataSequence` a `DataTable` bylo potřeba zvážit, jakou organizaci zvolit pro reprezentaci seznamu položek nebo n-tic. V úvahu připadaly dvě – pole zapouzdřené pomocí `MyList` a fronta zapouzdřená v `MyQueue`. Implementovány byly nakonec obě. Testováním se ukázalo, že z hlediska rychlosti vyhodnocení na tom jsou obě přibližně stejně. Efektivita spojového seznamu, kterým je fronta realizována, pravděpodobně nebyla zcela využita tím, že je pro každý uzel nutné instanciovat zaobalující třídu. Seznam není možné reprezentovat pomocnými proměnnými přímo v položce nebo n-tici, jelikož v rámci více rozpočítaných mezivýsledků může být položka nebo n-tice v jednu chvíli součástí více sekvencí nebo tabulek.

4.4.4 DOM

Z hlediska manipulace s XML je nejvýhodnější jeho reprezentace ve formě DOM (Document Object Model). DOM tvoří nedílnou součást datového modelu počínaje třídou `XmlNode` (viz. třídní diagram na obrázku 17). Součástí jsou dále třídy `XmlDocument`, `XmlAttribute`, `XmlText` a `XmlComment`, jejichž význam by měl být zřejmý.

Třída `XmlNode` představující abstraktní úroveň pro každý XML uzel poskytuje metodu `navigate`, která podle argumentu osy volá jednu z vnitřních metod pro navigaci. Samotná třída `XmlNode` implementuje metody pro navigaci na osy *ancestor*, *ancestor-or-self*, *descendant-or-self*, *following*, *following-sibling*, *preceding* a *preceding-sibling*. Navigace na *attribute*, *child* a *descendant* je záležitostí až třídy `XmlElement`, jelikož pro vyhodnocení těchto os je potřeba pracovat s přímými nebo nepřímými potomky [1].

4.4.5 Symbolické názvy proměnných

V kapitole 3.3.2, kde byl popisován překlad FLWOR výrazu, byla nadefinována nutnost substituce přístupu k proměnným pomocí operátoru Var na přístup do příslušné složky kontextové n-tice operátorem TupleAccess. Jednou z posledních úprav procesoru bylo zavedení symbolických názvů proměnných, resp. symbolických označení složek n-tic.

N-tic se totiž v případě složitějších dotazů může vygenerovat obrovské množství (ve složitých dotazech řádově milióny) a z hlediska definice si každá musí nést označení svých složek. Označení ve formě textového řetězce je sice přirozené, ale také zbytečně prostorově náročné. Navíc operátor Tuple, který n-tice generuje a přiřazuje jim názvy složek musí pro každou novou n-tici název kopírovat, což zhoršuje výkonnost procesoru.

Proto byla do třídy `DataModelManager` zaimplementována jednoduchá tabulka s dvojicemi *klíč* = *symbolický název* a metoda `int getSymbolicKey(string key)`, která pro zadaný klíč ve formě textového řetězce vrátí odpovídající symbolický název v podobě celočíselné hodnoty. Jestliže je metoda pro určitý klíč volána poprvé, postará se o vytvoření nového symbolického názvu a zavedení záznamu do tabulky, jinak vrátí odpovídající existující symbolický název.

Názvy složek n-tic jsou tedy tvořeny celými čísly. V plánu vykonávání dotazu se v operátorech Tuple a TupleAccess skutečné názvy přeloží na symbolické během statického vyhodnocování (viz. později v kapitole 4.8.1).

4.5 Parser

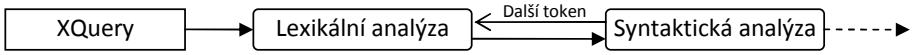
Implementace parseru vycházela z gramatiky definované standardem W3C ve formě EBNF [5]. Jediným omezením je to, že v současné době implementace nepracuje s unicode. Přestože procesor některé konstrukce XQuery nepodporuje, parsování vstupního dotazu je až na zmiňovaný unicode kompletní.

Existují v zásadě dvě možnosti, jak přistoupit k vytvoření parseru. První a jednodušší možnost je využití generátoru, který na vstupu obdrží gramatiku a vygeneruje zdrojový kód. Druhou možností je ruční implementace. Vzhledem k obtížné dostupnosti generátoru, který by byl schopen vytvořit XQuery parser v jazyce C++, byla nakonec využita druhá možnost, čili ruční im-

[1] Navigace na *descendant-or-self* je implementována ve třídě `XmlNode` přestože obecný uzel potomky mít nemusí. Tato navigace ale využívá navigaci na *descendant*, která je skutečně implementovaná až v `XmlElement`, pouze k ní přidává vlastní uzel.

plementace. Jak ale bude později popsáno, byla vytvořena speciální utilita, která základní skelet parseru vygenerovala.

Parsování se obvykle skládá ze dvou nezávislých kroků – lexikální a syntaktické analýzy.



Obrázek 19: Lexikální a syntaktická analýza (viz. [8])

Během lexikální analýzy je vstupní soubor převeden na posloupnost tokenů, na základě kterých syntaktická analýza vytvoří strom. Není nutné, aby tyto dva kroky probíhaly zcela odděleně, tzn. nejprve vytvoření pole tokenů a až poté provedení analýzy. Syntaktická analýza si postupně sama žádá o další a další tokeny. Blok provádějící lexikální analýzu se obvykle označuje jako tzv. *scanner*.

V případě XQuery je situace o něco málo komplikovanější a to především kvůli přímým konstruktorům. Prakticky to znamená, že se v XQuery mísí samotná konstrukce dotazu s jazykem XML. Zásadní rozdíl je jednak v přístupu k tzv. bílým znakům a jednak v pohledu na klíčová slova.

4.5.1 Terminální symboly

Je nutné rozlišovat *terminální symbol* a *typ terminálního symbolu*. Z ukázky v tabulce 8 vyplývá, že v určitých případech bude obsah terminálního symbolu rovnou dán jeho typem a v určitých případech bude potřeba konkrétní obsah upřesnit konkrétní hodnotou.

Terminální symboly bez obsahu	Terminální symboly s obsahem
TT_DOLLAR	TT_STRING_LITERAL
TT_NOT_EQUAL	TT_INTEGER_LITERAL
TT_DOT	TT_NCNAME
TT_COMMA	TT_DECIMAL_LITERAL
TT_LEFT_BRACKET	

Tabulka 8: Příklady terminálních symbolů bez obsahu a s obsahem

Terminální symboly se dělí na dvě skupiny (vyplývá ze standardu W3C). První skupinu tvoří tzv. *delimiting terminal symbols*, česky *oddělující terminály*. To jsou takové terminální symboly, které mohou následovat bezprostředně za sebou bez jakéhokoli oddělovače. Jedná se např. o pomlčku, závorku, znaménko plus, ukončovací značku tagu a další. Kompletní seznam je možné nalézt ve specifikaci standardu XQuery [5]. Pro lepší pochopení není složité představit si jednoduchý aritmetický výraz, kde za levou závorkou následuje ihned unární minus, tzn. pomlčka.

Druhou skupinou jsou tzv. *non-delimiting terminal symbols*, neboli *nedělící terminály*. To jsou především klíčová slova a identifikátory. Ty musí být mezi sebou odděleny bílou mezerou nebo nějakým jiným oddělujícím terminálem. Důvod je celkem zřejmý. Stačí si představit dvě klíčová slova bez jakéhokoli oddělení. V takovém případě by samozřejmě nebylo možné určit, zda se

jedná o dvě klíčová slova nebo jeden dlouhý identifikátor. Kompletní seznam těchto symbolů je opět součástí přílohy.

Nejednoznačnost terminálů

Gramatika XQuery nedefinuje terminální symboly úplně jednoznačně. To znamená, že určitý textový řetězec může být za různých okolností identifikován jako různý terminální symbol.

Jako příklad můžeme uvést *QName* a *NCName*, viz. ukázka 10. Oba symboly jsou v gramatice definovány jako terminální. Jak se má scanner zachovat, jestliže následujícím řetězcem na vstupu bude „`aaa:bbb`“? V tuto chvíli nelze s určitostí říct, zda scanner při požadavku na další terminál vrátí „`aaa`“ jako *NCName* nebo „`aaa:bbb`“ jako celý *QName*.

```
QName           ::= PrefixedName | UnprefixedName
PrefixedName    ::= Prefix ':' LocalPart
UnprefixedName  ::= LocalPart
Prefix          ::= NCName
LocalPart       ::= NCName
NCName          ::= Name - (Char* ':' Char*)
Name            ::= NameStartChar (NameChar)*
```

Ukázka 10: Problematické terminální symboly

Řešení problému spočívá v tom, že si blok syntaktické analýzy podle parsovaného gramatického pravidla sám určí množinu terminálních symbolů, kterou bude v danou chvíli ochoten akceptovat. To znamená, že syntaktická analýza postupně dotazuje scanner na terminální symboly, jejichž výskyt je v danou chvíli očekáván a scanner pouze odpovídá logickou hodnotou, zda daný terminální symbol aktuálně je či není na vstupu.

4.5.2 Syntaktický strom

Obvykle probíhá překlad formálního jazyka na operátory nebo jiná elementární volání (např. instrukce assembleru) rovnou během fáze parsování. V této implementaci jsou však obě fáze striktně odděleny a samotný překlad probíhá až na základě paměťové konstrukce syntaktického stromu.

Syntaktický strom je tvořen gramatickými pravidly použitými ve vstupním dotazu, kde vnitřní uzly tvoří neterminální symboly (gramatická pravidla) a listové uzly tvoří terminály. Kořenovým pravidlem je vždy pravidlo *Module* (viz. [5], příloha A.1).

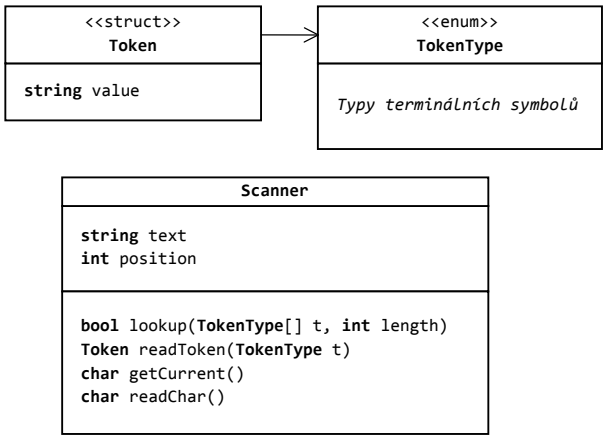
Striktní oddělení fáze parsování a překladu má několik výhod:

1. Oba bloky je možné nezávisle upravovat nebo v případě potřeby úplně vyměnit.

2. Reprezentace ve formě syntaktického stromu umožňuje provést určitou transformaci vstupního dotazu. Pokud by se práce dále rozšiřovala a z hlediska efektivity by se ukázalo, že normalizace vstupního dotazu bude nutná, může se provést právě na úrovni syntaktického stromu. V práci [9] je syntaktický strom používán jako model dotazu.

3. V souvislosti s případným propojením procesoru s indexovanou XML databází a vyhledáváním TPQ (Tree Pattern Query) ve stromu operátorů bude pravděpodobně nutné provést transformaci na tzv. TPNF [14].

4.5.3 Scanner



Obrázek 20: Třídní diagram scanneru

Funkci scanneru, čili lexikální analýzy, zajišťuje třída `Scanner`. Klíčovými metodami jsou `readToken`, `lookup`, `readChar` a `getCurrent`.

Implementace scanneru

Scanner (viz. třídní diagram na obr. 20) je instanciován vždy na základě nějakého vstupního textového řetězce text a pamatuje si pozici aktuálně čteného znaku v proměnné `position`. Poziční proměnná je inicializována na nulovou hodnotu. Metodou `lookup` je možné dotazovat scanner, zda od aktuální pozice přečíst danou lze sekvenci typů terminálních symbolů. Při tomto nahlížení se z hlediska uživatele scanneru nesmí nic změnit. Scanner sice během `lookup` s poziční proměnnou pracuje, avšak před navrácením výsledku dojde k obnovení jejího původního stavu.

Nahlížení je důležité proto, že ne vždy je možné rozhodnout o následujícím gramatickém pravidle na základě prvního přečteného terminálu. Gramatika tedy není typu LL1. Například v případě deklarací se konkrétní syntaktické pravidlo určí až tím, co následuje za klíčovým slovem `declare` (viz. pravidla na ukázce 11).

```
 [7] Setter               ::= BoundarySpaceDecl | DefaultCollationDecl |
                              BaseURIDecl |  ConstructionDecl |
                              OrderingModeDecl | EmptyOrderDecl |
                              CopyNamespacesDecl

[11] BoundarySpaceDecl    ::= "declare" "boundary-space" ("preserve" |
                              "strip")
```

```
[12] DefaultNamespaceDecl  ::= "declare" "default" ("element" | "function")
                               "namespace" URILiteral
[13] OptionDecl            ::= "declare" "option" QName StringLiteral
[14] OrderingModeDecl      ::= "declare" "ordering" ("ordered" | "unordered")
[15] EmptyOrderDecl        ::= "declare" "default" "order" "empty"
                                ("greatest" | "least")
[16] CopyNamespacesDecl    ::= "declare" "copy-namespaces" PreserveMode ","
                                InheritMode
```

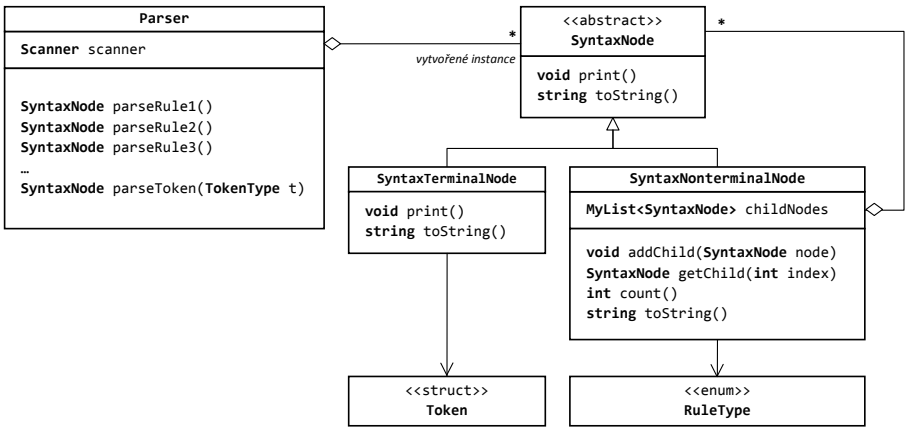
Ukázka 11: Gramatická pravidla nerozhodnutelná na základě prvního terminálu

Metoda `readToken` funguje podobně jako `lookup`, avšak její činnost končí navrácením obsahu jednoho typu terminálního symbolu (předaného argumentem) a úpravou `position`. Parser obvykle pomocí `lookup` nejprve vyzkouší několik možností a až poté se rozhodne, jakým způsobem se zanoří.

Velmi důležité veřejné metody jsou také `getCurrent` a `readChar`. Obě metody vrací znak vstupního řetězce na aktuální pozici `position`. Metoda `readChar` navíc před navrácením výsledku poziční proměnnou inkrementuje. Je zde možné nalézt analogii s metodami `lookup` a `readToken`.

4.5.4 Syntaktická analýza

Syntaktická analýza se provádí pomocí rekurzivního sestupu dle dané gramatiky. Z třídního diagramu na obrázku 21 je patrné, že pro implementaci syntaktického stromu byl využit návrhový vzor kompozit. Terminální uzel (`SyntaxTerminalNode`) se vždy odkazuje na konkrétní terminální symbol (`Token` – viz. třídní diagram na obr. 20). Neterminální uzel byl vytvořen vždy podle nějakého gramatického pravidla a odkazuje se tedy na jeden z prvků výčtu `RuleType`.



Obrázek 21: Třídní diagram parseru

Generování parseru

V úvodu podkapitoly 4.5 bylo naznačeno, že implementace parseru proběhla částečně automatizovaně. Vůbec první krok, ještě před zahájením práce na procesoru, spočíval ve vytvoření jednoduchého generátoru, který z dané gramatiky v EBNF základ parseru XQuery v jazyce C++ vytvořil. Jen pro ilustraci - metod `parse*` se ve třídě Parser nachází více než 100. Bez tohoto zjednodušení by bylo manuální kódování časově velmi náročné.

Tento jednoduchý generátor si poradil se základními konstrukcemi samotné gramatiky – sekvencemi, alternativami, iteracemi symbolů s povinným nebo nepovinným výskytem, výčty znaků atd. Komplikované bylo především sestavení kódu pro analýzu alternativy právě z důvodu nejednoznačnosti terminálů.

Výsledkem byl kód, jenž obsahoval metody `parse*`, které se za pomocí nahlížení navzájem volaly. Generátor velmi dobře posloužil pro vytvoření kódu analyzujícího XQuery dotazy bez přímých XML konstruktorů. Jejich parsování bylo nutné doimplementovat manuálně.

Ne vždy generátor sestavil kód u alternativ úplně správně tak, aby parser „nezabloudil“. Dodatečné úpravy byly v principu jednoduché, avšak vzhledem k množství gramatických pravidel časově dosti náročné. V mnoha případech bylo nutné upravit pořadí podmínek tak, aby např. bylo upřednostněno klíčové slovo oproti identifikátoru.

Ošetření chyb a správa paměti

Lexikální i syntaktická analýza může v případě nekorektně zadaného XQuery výrazu skončit chybou. Může se jednat např. o neočekávaný token, neuzavřenou závorku, neukončený tag a další. Jestliže parser narazí na chybu, vygeneruje se výjimka s příslušným chybovým kódem.

Součástí třídy `Parser` (viz. třídní diagram na obr. 21) je datová struktura, která postupně shromažďuje reference na vzniklé uzly syntaktického stromu. Tato struktura je v současné verzi řešena pomocí `MyList<SyntaxNode>`. Vždy při instanciování terminálního nebo neterminálního uzlu se do struktury okamžitě přidá vzniklá reference. Všechny vzniklé instance je tak možné i v případě chyby a následně generované výjimky řádně odalokovat.

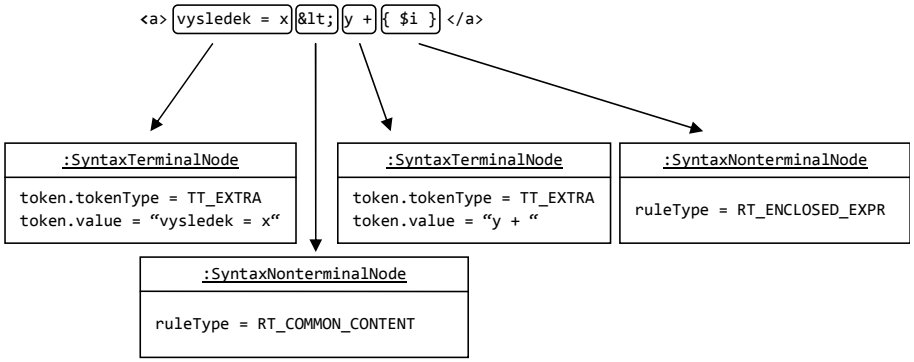
Parsování XML

Vypočtené konstruktory jsou z hlediska parsování rutinní záležitost. Menší komplikace nastanou při používání konstruktorů přímých. Jakmile parser narazí na pravidlo *DirectConstructor*, nežádá scanner dále o tokeny, ale čte vstupní soubor po znacích pomocí metod `readChar` a `getCurrent`, které jsou součástí scanneru (viz. obr. 20).

Během parsování XML je potřeba sledovat začátky a konce elementů, komentářů nebo procesních instrukcí. V hlavičkách uzlů se mohou samozřejmě objevit také atributy. Kromě toho je potřeba počítat s předdefinovanými entitami např. pro zápis znaku „<“ nebo „>“, tzn. `<` nebo `>`. Nakonec se v obsahu elementů, atributů nebo komentářů může objevit i zanořený výraz ve složených závorkách.

Byl vytvořen speciální typ tokenu `TT_EXTRA`, který slouží právě pro účely, kdy se tokeny nevytváří během lexikální, ale až během syntaktické analýzy. Na obrázku 22 je naznačeno parsování krátkého konstruktoru XML elementu *a*.

Celkem zajímavou vlastností jazyka XQuery je, že na celý XQuery procesor lze svým způsobem pohlížet také jako na parser XML. Pokud se v XML vyhneme některým speciálním konstrukcím,



Obrázek 22: Princip parsování XML elementu

které jsou charakteristické pro XQuery, pak ono XML samo o sobě představuje platný XQuery dotaz. Tento „dotaz" se přeloží na operátory tak, že výsledkem vyhodnocení je DOM. Aby bylo jednoznačně rozlišeno, co je skutečně XML a co je XQuery dotaz, gramatika XQuery zakazuje, aby se v hlavičce dotazu nacházela procesní instrukce `<?xml ... ?>`.

Při testování procesoru během vývoje byla uvedená vlastnost uplatněna ve vestavěné funkci pro načítání vstupních XML dokumentů. Později byla funkce z hlediska efektivity upravena tak, aby pro parsování XML dokumentu využívala Xerces Parser.

4.6 Kompiler

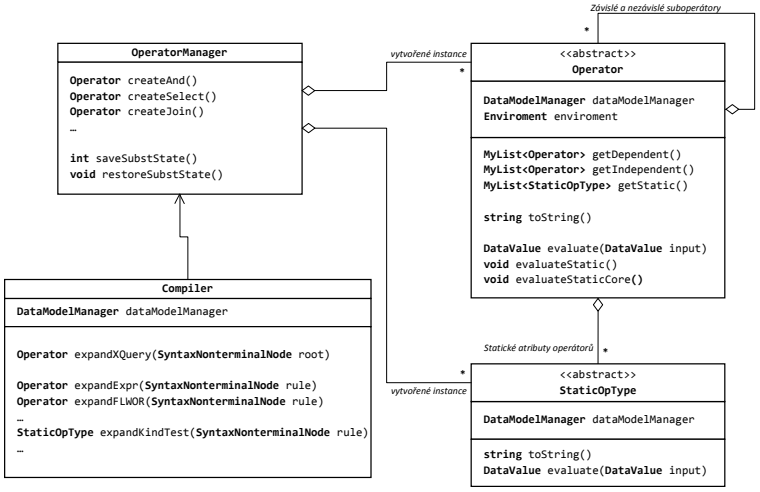
Jakmile máme připravený syntaktický strom, potřebujeme z něj vytvořit plán vykonávání dotazu. O tuto transformaci se stará kompiler neboli překladač implementovaný třídou `Compiler`. Třídní diagram základních typů překladače se nachází na obr. 23.

Ze stejných důvodů, jako u syntaktické analýzy, je při vytváření instancí konkrétních operátorů nutné pečlivě udržovat seznam vzniklých referencí. O jejich evidenci se v tomto případě nestará sama třída `Compiler`, ale oddělená třída `OperatorManager`. K tomuto rozdělení došlo proto, že kompiler není jediné místo, kde mohou objekty operátorů vznikat.

Na diagramu jsou dále zachyceny abstraktní třídy `Operator` a `StaticOpType` pro reprezentaci operátoru a statické složky operátoru.

4.6.1 Kompilace

V tomto kroku můžeme předpokládat, že byl syntaktický strom vytvořen pomocí parseru a je tedy v souladu s gramatikou XQuery. Součástí kompileru je množství metod `expand*`, které na základě nějaké konstrukce XQuery, tzn. syntaktického uzlu, sestaví operátor, resp. opět celý strom operátorů.



Obrázek 23: Třídní diagram překladače

Metoda expandExpr

Tato metoda třídy `Compiler` představuje univerzální výchozí bod pro kompilaci jakéhokoli gramatického pravidla, tzn. neterminálu. Na základě typu vstupního neterminálního uzlu se metoda rozhodne, zda je pravidlo natolik jednoduché, že se o překlad postará sama nebo pro zpracování zavolá jinou konkrétní metodu `expand*`.

Příklad 4: *Kompilace větvení*

Algoritmus 1 demonstruje kompilaci jednoduchého pravidla větvení *IfExpr* podle gramatického pravidla na ukázce 12. Zde je gramatikou přesně určeno, na jakých pozicích se poduzly představující podmínku, kladnou a zápornou větev nacházejí.

```
[45] IfExpr ::= "if" "(" Expr ")" "then" ExprSingle "else" ExprSingle
```

Ukázka 12: Gramatické pravidlo pro větvení

V tomto případě je vytvořena instance operátoru Cond. Instance není založena standardně klíčovým slovem `new`, ale metodou `createCond` (objektu správce operátorů `OperatorManager`), která podle argumentů rovnou nastaví v pořadí operátor pro výpočet kladné větve, operátor pro výpočet záporné větve a operátor pro výpočet podmínky. Z gramatiky víme, že se neterminální symbol pro logický výraz rozhodující podmínku nachází v rámci pravidla *IfExpr* pevně na indexu 2, pravidlo pro kladnou větev na indexu 5 a pravidlo pro zápornou větev na indexu 7. Pro kompilaci těchto pravidel se rekurzivně opět zavolá metoda `expandExpr`.

Algoritmus 1: Část metody `expandExpr` pro překlad podmínky

```
if překládaným pravidlem je IfExpr then
    return operatorManager.createCond(
      expandExpr(potomek syntaktického uzlu na indexu 5),
      expandExpr(potomek syntaktického uzlu na indexu 7),
      expandExpr(potomek syntaktického uzlu na indexu 2));
end
```

Volání jiné metody pro překlad

Jestliže univerzální metoda `expandExpr` zjistí, že překládá např. pravidlo *AndExpr* nebo *OrExpr*, není už kompilace tak přímočará jako v předchozím případě a zavolá se jednoúčelová metoda `expandAndExpr` nebo `expandOrExpr`. Kompilaci by sice v tomto případě ještě bylo možné vyřešit rovnou v těle `expandExpr`, avšak výsledný kód by se takto pomalu stal velmi nepřehledným.

Propadávání pravidel

Existuje mnoho případů, kdy se neterminál pouze alternativně přepisuje na jeden jiný neterminál.

```
NumericLiteral := IntegerLiteral | DecimalLiteral | DoubleLiteral
```

Ukázka 13: Propadající pravidlo

Z hlediska parsování jsou takovéto přepisy důležité, ale z hlediska kompilace tato pravidla svým způsobem propadávají a nedochází k vytváření žádného operátoru. Jestliže se tedy kompiluje např. pravidlo *NumericLiteral* z ukázky 13, pak metoda `expandSyntax` rovnou zavolá sama sebe s nultým potomkem neterminálního uzlu, který právě zpracovává. V tomto případě to bude vždy jedeno z pravidel *IntegerLiteral*, *DecimalLiteral* nebo *DoubleLiteral*. Teprve tato pravidla budou zkompilována na operátor Scalar.

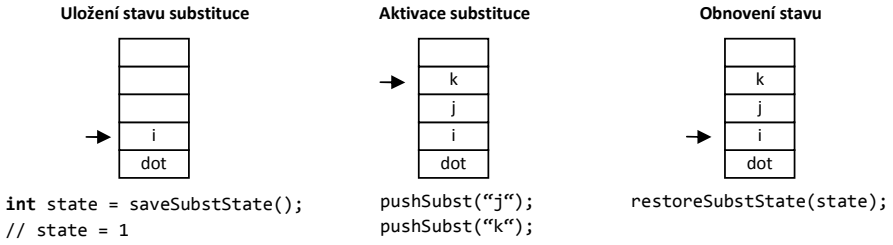
Substistuce proměnných

O substituci proměnných se stará již zmiňovaná třída `OperatorManager`, která primárně slouží pro správu paměti. Využívá se zde toho, že jediným možným místem v programu, kde může vzniknout operátor *Var* pro přístup k proměnné, je metoda `createVar` právě ve třídě `OperatorManager`. V této metodě lze projít tabulku aktuálně substituovaných proměnných a v případě nalezení shody vrátit místo instance *Var* instanci operátoru *TupeAccess* s přístupem do kontextové n-tice IN. Kdy se substituce provádí a jaký je její význam popisuje kapitola 3.3.2.

Z hlediska kompileru, čili třídy `Compiler` a některé z metod `expand*` (viz. třídní diagram na obr. 23), lze používání substitucí připodobnit zásobníku (proto také název metody `pushSubst`). Jestliže byla v pořadí aktivována nejprve substituce proměnné var_1 a až poté var_2, musí dojít ke zrušení substituce v opačném pořadí, tzn. var_2, poté var_1.

Duálně k metodě `pushSubst` by tedy měla existovat také nějaká metoda `popSubst`.

Ve skutečnosti je tomu ale trochu jinak. Zatímco aktivování substitucí probíhá obvykle postupně např. v určitých fázích kompilace FLWOR výrazu, zrušení substitucí se provádí najednou. Z toho důvodu je výhodnější celkový stav substitucí nejprve nějakým způsobem uložit, poté pokračovat v kompilaci, podle potřeby substituce jednotlivých proměnných postupně aktivovat a nakonec se k uloženému stavu opět vrátit. Uložení a obnovení substitučního stavu se provádí metodami `getSubstState` a `restoreSubstState`.



Obrázek 24: Princip substituce proměnných

Z ukázky na obrázku 24 je patrné, že při obnovení stavu substituce zůstávají v tabulce názvy proměnných. To samozřejmě ničemu nevadí, protože při testu na substituci probíhá vyhledávání pouze po aktuální index.

4.6.2 Kompilace složitějších konstrukcí

Mezi komplikovanější konstrukce XQuery na kompilaci patří FLWOR a XPath výrazy. V obou případech se však jedná pouze o opatrnou manipulaci s gramatickými pravidly a vytváření operátorů podle použité algebry, tzn pravidel nadefinovaných v kapitolách 3.3.2 a 3.3.3.

Z hlediska implementace je obtížná zejména správná lokalizace klíčových gramatických pravidel v syntaktickém stromu. Tato úloha by byla pravděpodobně snazší, kdyby překlad probíhal rovnou během parsování. Přišli bychom ale o určité výhody (viz. kapitola 4.5.2). Pro ilustraci jsou základní gramatická pravidla pro FLWOR a XPath výrazy uvedena v ukázkách 14 a 15.

```
[33]  FLWORExpr      ::= (ForClause | LetClause)+ WhereClause? OrderByClause?
                         "return" ExprSingle
[34]  ForClause      ::= "for" "$" VarName TypeDeclaration? PositionalVar? "in"
                         ExprSingle ("," "$" VarName TypeDeclaration?
                         PositionalVar? "in" ExprSingle)*
[35]  PositionalVar  ::= "at" "$" VarName
[36]  LetClause      ::= "let" "$" VarName TypeDeclaration? ":=" ExprSingle
                         ("," "$" VarName TypeDeclaration? ":=" ExprSingle)*
[37]  WhereClause    ::= "where" ExprSingle
```

Ukázka 14: Kořenové neterminály FLWOR

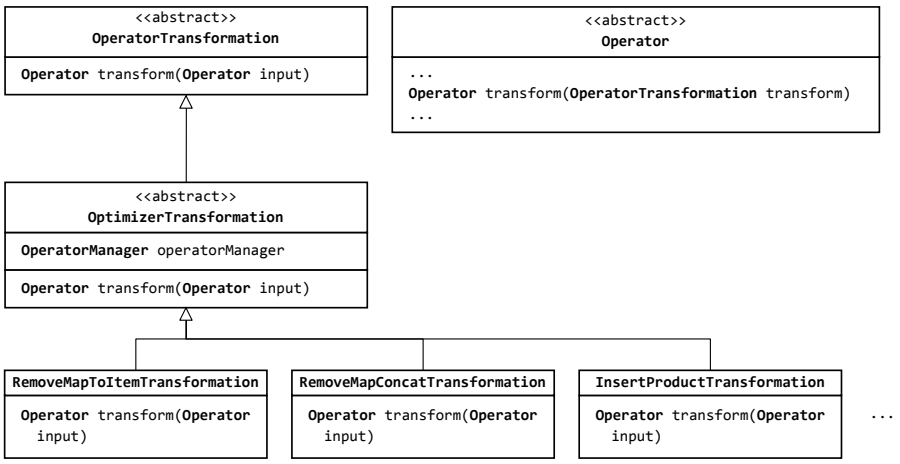
```
[68]  PathExpr          ::= ("/" RelativePathExpr?)
                          | ("//" RelativePathExpr)
                          | RelativePathExpr
[69]  RelativePathExpr  ::= StepExpr (("/" | "//") StepExpr)*

[70]  StepExpr          ::= FilterExpr | AxisStep
[71]  AxisStep          ::= (ReverseStep | ForwardStep) PredicateList
[72]  ForwardStep       ::= (ForwardAxis NodeTest) | AbbrevForwardStep
[75]  ReverseStep       ::= (ReverseAxis NodeTest) | AbbrevReverseStep
[82]  PredicateList     ::= Predicate*
```

Ukázka 15: Kořenové neterminály pro XPath

4.7 Optimalizace

Samotná optimalizace se skádá z postupného uplatnění několika optimalizačních pravidel, která byla nadefinována v kapitole 3.4. Aplikace každého z nich je nepovinná, resp. výsledek vyhodnocení dotazu musí s optimalizací i bez ní stejný. Optimalizátor funguje na modulárním principu a je tedy možné snadno určovat, jaká pravidla se skutečně uplatní a v jakém pořadí.



Obrázek 25: Třídní diagram optimalizace

Třída `OperatorTransformation` z třídního diagramu na obr. 25 obecně popisuje nějakou transformaci operátoru metodou `transform` bez návaznosti na optimalizátor. Konkrétní implementace této metody může zajistit nahrazení jednoho operátoru nebo kombinace jiným operátorem nebo kombinací.

Abstraktní třída `OptimizerTransformation` popisuje transformaci určenou výhradně pro optimalizátor. Tyto dvě úrovně abstrakce vznikly proto, aby byl celý systém schopen s minimálními úpravami fungovat i v teoretickém případě odebrání všeho, co se optimalizací týče. Vzhledem k tomu, že optimalizací vznikají nové operátory, je potřeba transformacím optimizéru předat také instanci `OperatorManager`.

Aby bylo možné transformaci propagovat na celý strom, došlo k rozšíření třídy `Operator` z třídního diagramu na obrázku 23 o abstraktní metodu `transform`. Každá konkrétní implementace operátoru musí v této metodě na základě argumentem předané transformace upravit nejprve všechny své suboperátory a poté vrátit transformovanou podobu sama sebe (`this`). Na algoritmu 2 je zachycena transformace operátoru Select. Identifikátor `op` představuje operátor pro výpočet nezávislého vstupu, `boolean` je operátor pro výpočet mezivýsledku – podmínky selekece.

Algoritmus 2: Transformace operátoru Select

input : transformace *t* ve formě instance `OperatorTransformation`
output: transformovaná podoba operátoru Select

op ← *op*.transform(*t*);
boolean ← *boolean*.transform(*t*);
return *t.transform(*this*)*;

Příklad 5: *Transformace vložení* Product

Pro ukázku zde bude na algoritmu 3 popsáno nahrazení operátoru MapConcat operátorem Product, tzn. kartézským součinem. Proč je toto možné a za jakých okolností, popisuje kapitola 3.4.3. Logiku transformace zajišťuje metoda třídy `InsertProductTransform`, která dědí z `OptimizerTransformation` na třídním diagramu (obr. 25).

Transformace se uplatní pouze tehdy, jestliže zpracovává operátor MapConcat, jinak metoda vrátí vstupní operátor beze změny. Při uplatnění transformace je potřeba zjistit, zda se v závislém podoperátoru nenachází reference na kontextovou n-tici, resp. operátor IN. Prohledat se musí celý podstrom, přičemž v podstromu se prohledávají pouze nezávislé suboperátory. Při přechodu na závislý suboperátor by se případně nalezený IN nevztahoval k MapConcat, pro který IN hledáme (viz. kapitola 3.2.3).

Jednou z možností, jak tuto detekci provést je pro průchod stromem ať už do hloubky nebo do šířky použít rekurzi. Rekurzivní procházení operátorů je nahrazeno zásobníkem, na jehož vrchol je ze začátku umístěn nezávislý podoperátor právě zpracovávaného *mapConcat*.

V případě, že má dojít k nahrazení, je místo původního *mapConcat* navrácena nová instance Product s přebraným závislým i nezávislým podoperátorem.

4.8 Vyhodnocování

Poslední fáze spočívá ve vyhodnocení připraveného plánu dotazu. V blokovém schématu na obrázku 15 je sice naznačeno, že se o vyhodnocení stará procesor. Ve skutečnosti je principem vyhodnocení volání metody operátoru `evaluate`, která argumentem přebírá kontextovou hodnotu. V další části textu budou popsány pouze některé složitější operátory. Většina operátorů provádí triviální operace, jejichž popis by zde postrádal smysl.

Algoritmus 3: Transformace vložení Product

input : operátor *op* – kořen transformovaného podstromu
output: transformovaná podoba

if *transformovaný operátor je* MapConcat **then**
  inicializuj prázdný zásobník ;
  mapConcat ← přetypuj *op* na MapConcat ;
  vlož do zásobníku závislý operátor z *mapConcat* ;
  while *zásobník není prázdný* **do**
    op_{pom} ← vyjmi prvek ze zásobníku ;
    if op_{pom} *je operátor* IN **then**
      byl nalezen závislý IN ;
      ukonči cyklus ;
    else
      vlož do zásobníku všechny nezávislé operátory z op_{pom} ;
    end
  end
  if *byl nalezen závislý* IN **then**
    return *instanciuj* Product *s původním nezávislým a závislým vstupem z mapConcat* ;
  end
end
return *vstupní operátor op beze změny* ;

4.8.1 Vyhodnocení statických složek operátorů

V případě některých operátorů je před spuštěním vykonávání plánu potřeba nejprve inicializovat nějaké vnitřní proměnné na základě statických složek. Jedná se např. o překlad názvů složek n-tic na symbolické názvy (viz. kapitola 4.4.5).

O statické vyhodnocení se starají metody `evaluateStatic` a `evaluateStaticCore` (viz. třídní diagram na obr. 23). První metoda pouze zajišťuje rekurzivní volání sama sebe na závislé i nezávislé operátory a volání `evaluateStaticCore`. Ve druhé uvedené metodě je možné podle potřeby provést inicializaci, která závisí na statických složkách a složení závislých nebo nezávislých suboperátorů. Voláním `evaluateStatic` na kořenový operátor celého stromu ještě před začátkem vyhodnocování dotazu tak můžeme přednastavit některé vnitřní vlastnosti operátorů, které jsou během vyhodnocování neměnné.

4.8.2 Operátor Select

Činnost algoritmu 4 pro vyhodnocení Select není sama o sobě nijak zajímavá, cílem ukázky pseudokódu 4 je prezentace všech předem připravených vlastností datového modelu.

Nejprve je vyhodnocen nezávislý podoperátor *op* nad stejným kontextem, s jakým byl volán právě zpracovávaný Select. Jedná se o operátor vracející vstupní tabulku n-tic. Dále si lze všimnout jednak praktického využití metody pro přetypování `asDataTupleEnumerable` a jednak toho, že na vyhodnocený vstup není pohlíženo jako na tabulku, ale jako na `DataTupleEnumerable`. Nezáleží na tom, zda je vstupem celá tabulka, nebo jedna n-tice.

Algoritmus 4: Vyhodnocení operátoru Select

```
input  : kontextová hodnota in
output: tabulka n-tic po provedení selekce

DataValue inputValue ← op.evaluate(in) ;
DataTupleEnumerable inputEnumerable ← inputValue.asDataTupleEnumerable() ;
DataTable outputTable ← dataModelManager.createDataTable() ;
TableIterator iterator ← inputEnumerable.getIterator() ;

while iterator.hasNext() do
    DataTuple tuple ← iterator.getNext() ;
    dataModelManager.pushInstanceCounters() ;
    DataValue value ← boolean.evaluate(tuple) ;
    dataModelManager.popInstanceCounters() ;
    BooleanItem boolItem ← value.asBooleanItem() ;
    if boolItem.value = true then
        přidej n-tici tuple do výstupní tabulky outputTable ;
    end
end

return outputTable;
```

Pomocí `dataModelManager` je vytvořena instance výstupní tabulky a může se zahájit iterace přes vstup. Pro každou zpracovávanou n-tici je vyhodnocen závislý `boolean`, který vyhodnocuje podmínku selekce.

Před resp. po vyhodnocení `boolean` jsou volány metody `pushInstanceCounters` a `popInstanceCounters`, které zajistí možnost recyklace instancí proměnných. Ze všech hodnot, které potenciálně vzniknou během zpracování `boolean` nás totiž zajímá pouze výstupní `BooleanItem` a i tu je možné po přečtení skutečné logické hodnoty okamžitě zahodit. Ostatní hodnoty, musely vzniknout z důvodu nějakého mezivýsledku, který nás už dále nezajímá. Problém by mohl vzniknout jen v případě používání globálních proměnných, které ale zatím implementovány nejsou. Pokud by během výpočtu mezivýsledku došlo k vytvoření obsahu globální proměnné, nesměl by být tento obsah zahozen.

4.8.3 Operátor TreeJoin

Operátor *TreeJoin* je jedním z těch, které operují pouze nad sekvencemi položek. Nejsložitější část činnosti operátoru je navigace v XML stromu, kterou neřeší sám operátor, ale třída `XmlNode`.

Navigace na osu descendant

Nejen, že je navigace na všechny přímé nebo nepřímé potomky často využívaná přímo v XQuery dotazu, pomocí této navigace lze také rozepsat některé další – `preceding`, `following` nebo `descendant-or-self`.

Algoritmus pro zjištění všech potomků v DOM reprezentaci je sám o sobě velmi jednoduchý. Stačí rekurzivně projít celý podstrom pro daný XML element. Je pouze potřeba zachovat pořadí

uzlů, což je zajištěno procházením do hloubky. Byly otestovány celkem čtyři možnosti implementace této metody – jednoduchá rekurze, přepis na nerekurzivní podobu pomocí zásobníku realizovaného spojovým seznamem, přepis na nerekurzivní podobu pomocí zásobníku realizovaného polem a průchod předpřipraveným spojovým seznamem, který byl konstruován během vytváření DOM (viz. níže).

Poslední zmiňovaný způsob se experimentálně ukázal jako nejefektivnější, nicméně nedodržuje původní pořadí uzlů podle dokumentu. Druhým nejrychlejším se ukázala klasická rekurze, o něco pomalejší byla nerekurzivní varianta se zásobníkem realizovaným pomocí `MyList` a nejhůře dopadla nerekurzivní varianta se zásobníkem realizovaným spojovým seznamem.

Vše bylo testováno na na XML dokumentu o velikosti 18 MB s cca 300 tisíci XML uzly (viz. popis v kapitole 5) a jednoduchém XPath dotazu `//item[@id="item4016"]`. Průměrné časy vyhodnocování jsou uvedeny v tabulce 9.

Průchod spojovým seznamem	110 ms
Použití rekurze	120 ms
Nerekurzivní varianta pomocí pole	135 ms
Nerekurzivní varianta pomocí spojového seznamu	210 ms

Tabulka 9: Srovnání algoritmů pro navigaci na osu *descendant*

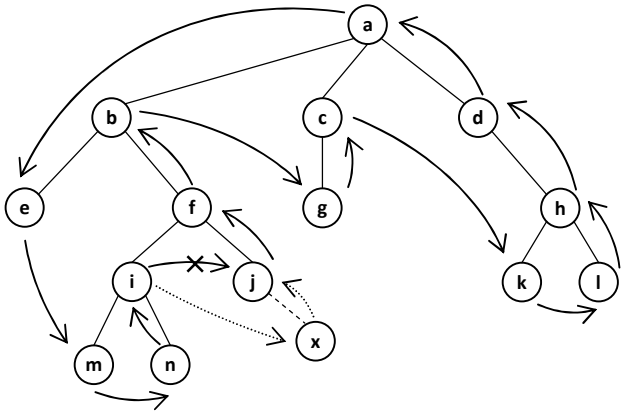
Efektivitu rekurze lze v tomto případě zdůvodnit tím, že se v těle metody fyzicky používají pouze dvě lokální proměnné. Argumenty metody jsou označeny jako `const`, takže při rekurzivním volání by nemělo docházet ani k vytváření lokálních kopií. Navíc u běžných XML se úroveň zanoření pohybuje maximálně v řádu jednotek až desítek.

Neefektivitu zásobníku realizovaného spojovým seznamem lze vysvětlit tím, že pro každou položku je nutné vytvořit novou instanci zaobalujícího uzlu.

Postorder seznam

Relativně zajímavá experimentální metoda podávající mírně lepší výsledky než rekurze je použití předpřipraveného spojového seznamu, který lze velmi jednoduše konstruovat při vytváření DOM a reprezentovat několika pomocnými proměnnými přímo ve třídě `XmlNode`. Seznam je postupně vytvářen tak, aby průchodem vznikl postorder zápis. Každý nově přidaný uzel je zařazen bezprostředně před svého rodiče. Procházení všech potomků začíná nalezením nejlevějšího uzlu celé větve. Poté je seznam procházen tak dlouho, než je nalezen pro danou větev kořenový uzel. Průchod sice nevrací uzly podle původního uspořádání v dokumentu, což ale nevadí, pokud bude procesor ve výchozím stavu pracovat v tzv. *unordered* režimu (viz. [5]). Předpokladem je, že procesor nikdy nekonstruuje DOM tak, že by uzly např. vkládal na určitý index – uzel je vložen do svého rodiče vždy jako poslední.

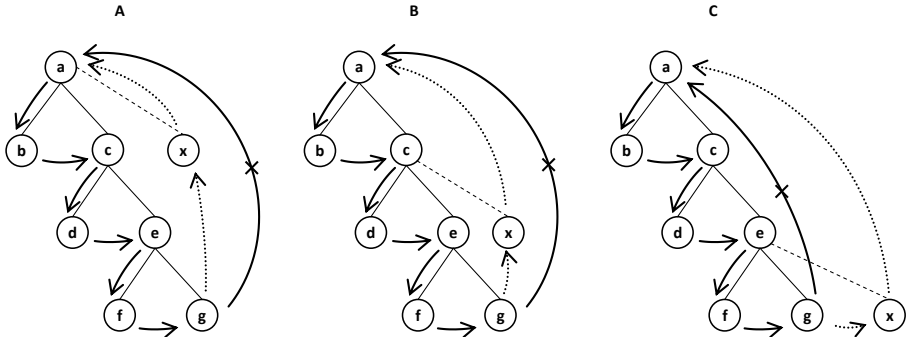
Vkládání do stromu s postorder zápisem (viz. obr. 26) probíhá jednoduše tak, že přerušíme „příchozí“ vazbu rodiče a protáhneme ji přes nově vložený uzel. Pro seznam to znamená, že nově vložený uzel bude zařazen před svého rodiče. Aby bylo možné z uzlu j okamžitě nalézt uzel i, kde je potřeba vazbu přerušit, je seznam ve skutečnosti implementován jako obousměrný.



Obrázek 26: Postorder seznam

Vytvoření takového seznamu, který by průchodem dával klasický preorder zápis a tím vracel uzly podle pořadí v dokumentu, možné je. Vyhledávání pak probíhá s rozdílem, že uzly navštěvujeme v pořadí od kořene až po nejpravějšího potomka dané větve. Problém je ale v průběžné konstrukci takového seznamu.

Na obrázku 27 jsou znázorněny tři rozdílné situace při vkládání uzlu *x* do vznikající stromové DOM reprezentace. Aby zůstal zachován preorder zápis, musí být ve všech třech případech (A, B, C) přerušena vazba mezi uzly *g* a *a*. V případě A by se mohlo zdát, že jde o tu vazbu, která je „příchozí" do rodičovského uzlu, kam se *x* vkládá. Naopak případ C by mohl naznačovat na pouhé přesměrování z posledního přímého potomka rodičovského uzlu. Obecně by mechanizmus musel pracovat tak, že se nejprve nalezne nejpravější potomek ve větvi toho elementu, kam se *x* vkládá, a tam se provede přesměrování. To, co bychom tedy získali při provádění navigace na *descendant*, bychom pak ztratili používáním konstruktorů XML elementů, jelikož každým novým vložením uzlu do vznikajícího stromu by muselo proběhnout vyhledání nejpravějšího potomka.



Obrázek 27: Preorder seznam

Složitost všech uvedených algoritmů se pohybuje v $\theta(n)$, kde n představuje počet uzlů, které musí algoritmus navštívit. Nejnákladnější operací je v tomto případě test uzlu (test na jméno, test na atribut a další). Efektivita jednotlivých algoritmů souvisí s režií kolem datových struktur, na základě kterých algoritmus pracuje.

TreeJoin

Vraťme se tedy k činnosti samotného operátoru TreeJoin, která znázorněna algoritmem 5.

Algoritmus 5: Implementace TreeJoin

```
input  : kontextová hodnota in
output: sekvence s uzly po provedení navigace

DataValue inputValue ← independentOp.evaluate(in) ;
DataItemEnumerable enumerable ← inputValue.asDataItemEnumerable() ;
SequenceIterator iterator := enumerable.getIterator();

inicializuj prázdný seznam MyList⟨XmlElement⟩ items;

while iterator.hasNext() do
    XmlNode node ← iterator.getNext() ;
    node.navigate(axis, nodeTest, items) ;
end
DataSequence outputSequence ← dataModelManager.createDataSequence() ;

// Eliminace duplicit

inicializuj prázdný MyHashSet⟨XmlElement⟩ hashSet ;

for všechny položky item v items do
    if hashSet neobsahuje položku item then
        přidej položku item do výstupní sekvence outputSequence ;
        přidej položku item do hash tabulky hashSet.
    end
end

return outputSequence ;
```

Nejprve je potřeba vyhodnotit nezávislý vstup *independentOp*, který je následně přetypován na `DataItemEnumerable`. Opět tedy zakrytí rozdílu mezi sekvencí a položkou. Iterátorem jsou procházeny všechny položky vstupní sekvence a metodou navigate postupně nashromážděny do seznamu `items`.

V podstatě jediným složitějším úkolem samotného TreeJoin je eliminace duplicit. Nejjednodušší implementací by bylo použití zanořené smyčky, což ale z hlediska efektivity určitě není nejlepší řešení. Právě z toho důvodu byla implementována množina `MyHash`, která je součástí univerzálních struktur popsaných výše v kapitole 4.3.

4.8.4 **Operátor** IN

Z hlediska správného pochopení, jak funguje předávání hodnoty kontextu je vhodné vysvětlit skutečnou funkci operátoru IN. Úkolem tohoto operátoru není nic jiného, než navrácení kontextu, se kterým bylo voláno jeho vyhodnocení. Triviální postup je uveden na algoritmu 6. Ve většině případů je obsahem IN n-tice, pouze v případě závislého vstupu operátoru MapFromItem jde o položku.

Algoritmus 6: Implementace IN

input : kontextová hodnota *in*
output: kontextová hodnota *in*

return *in* ;

4.8.5 **Operátor** Call

Relativně zajímavý operátor z hlediska implementace představuje Call, čili volání funkce. Výčet všech základních vestavěných funkcí byl uveden v tabulce 5. Pro účely testování fungovala první verze vyhodnocování v principu velmi jednoduše formou jedné globální statické metody, viz algoritmus 7.

Algoritmus 7: Původní implementace Call

input : název funkce, obsah kontextu IN, pole argumentů funkce
output: výsledek funkce

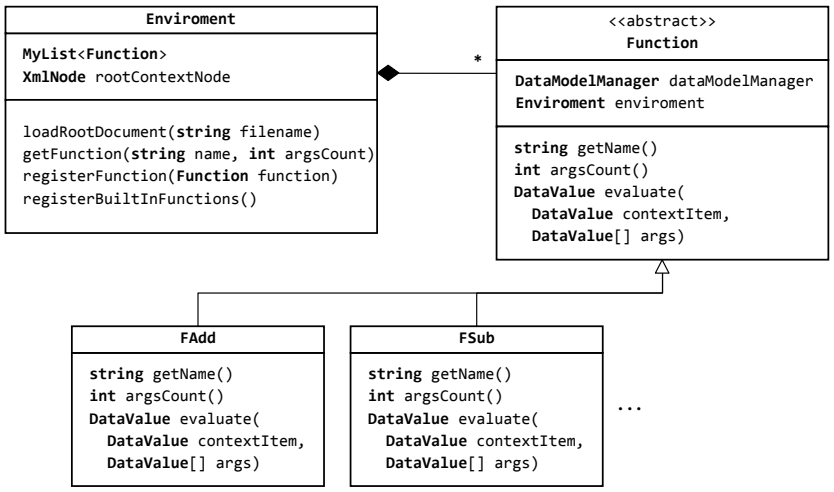
if *název funkce je „add“* **then**
| zpracování funkce add
else if *název funkce je „sub“* **then**
| zpracování funkce sub
else if *název funkce je „mul“* **then**
| zpracování funkce mul
end
...

Toto naivní řešení s sebou neslo několik významných nevýhod:

- nutnost zásahu do operátoru Call vždy při implementaci další funkce,
- ad-hoc řešení – chybějící modulární stavba,
- nutnost několikanásobného porovnávání řetězce názvu funkce při každém požadavku na vyhodnocení.

Prostředí

Nový princip modulární stavby funkcí je znázorněn na obrázku 28. Třída `Enviroment` slouží pro popis prostředí, ve kterém bude XQuery dotaz vykonáván. Prostředím se rozumí to, jaké funkce



Obrázek 28: Třídní diagram modulární stavby funkcí

budou k dispozici, co je kořenový uzel a výhledově také jaké jsou např. globální proměnné nebo konstanty. Dle specifikace W3C by se dalo říct, že právě tato třída popisuje tzv. *statický kontext*.

Z diagramu by mělo být evidentní, jakým způsobem modulární řešení funguje. Pro funkci je vytvořena abstraktní třída `Function`, každá konkrétní funkce pak musí umět vrátit svou signaturu, tzn. název a počet vstupních argumentů. Do budoucna by součástí signatury mohly být také jejich datové typy. Před samotným používáním funkce musí proběhnout její registrace do prostředí metodou `registerFunction`.

Během vyhodnocování statických složek operátoru Call (kap. 4.8.1) je dáno, jaký bude mít volaná funkce název, a počtem podoperátorů dáno, kolik argumentů bude funkce potřebovat. Z prostředí je tedy možné jednoduše získat odkaz na potřebnou funkci včetně její implementace ještě před zahájením vyhodnocování. Dojde tedy k „prolinkování“ mezi konkrétním operátorem Call a algoritmem samotné funkce.

Zavedením modulární stavby funkcí skutečně došlo ke zrychlení vykonávání dotazu na ukázce 16. Na zkušebním XML dokumentu (viz. níže kap. 5) došlo ke zlepšení z přibližně 330 na 290 ms díky efektivnějšímu volání funkce `eq` pro porovnávání.

```
for $i in //item
where some $t in $i//listitem/text() satisfies $t = "Test"
return $i
```

Ukázka 16: Testovací dotaz

4.8.6 Další operátory

Mezi další mírně komplikovanější operátory patří např. MapConcat, Join nebo Product. U těchto velmi podobných operátorů snad stojí za zmínku akorát to, že spojování n-tic se provádí na rela-

tivně nízké úrovni. Místo postupného překopírování názvů a hodnot z jedné n-tice do druhé se používá nízkoúrovňová kopie bloku paměti.

Složitějším operátorem byl dále OrderBy především kvůli nutnosti implementace QuickSort jako efektivního algoritmu třídění. Ten byl nakonec implementován mimo třídu samotného operátoru OrderBy pro možnost opakovaného využití třízení `MyList`.

Kromě všech výše popsaných principiálně důležitých tříd se ve skutečné implementaci objevuje velké množství dalších pomocných struktur, metod, proměnných, konverzních funkcí atd. Ty ale nejsou z hlediska popisu funkčnosti procesoru úplně podstatné. V případě, že by se práce dále rozšiřovala, bylo by potřeba vytvořit kompletní programátorskou dokumentaci. V tuto chvíli je samotný zdrojový kód relativně detailně okomentován a názvy všech identifikátorů by do určité míry měly být samopopisné. K udržování programátorské dokumentace by bylo možné využít např. nástroje Doxygen, který umožňuje automatické generování systému HTML stránek na základě komentářů a samotného kódu.

5 Experimentální vyhodnocování

Tato kapitola demonstruje na ukázkových XQuery dotazech naimplementovaný algebraický procesor. Všechny testy proběhly na PC s konfigurací uvedené v tabulce 10.

CPU	Intel Core 2 Duo E7300, 2.66 GHz
Operační paměť	4 GB
Operační systém	Microsoft Windows 7 Proffesional 64-bit

Tabulka 10: Konfigurace testovacího PC

Testovací XML dokument je zjednodušením dokumentu z testů XMark [17]. Základní parametry jsou uvedeny tabulce 11. Dokument je součástí CD přílohy této práce.

Velikost souboru	18 247 kB
Počet XML uzlů	313 296
Počet elementů	271 848
Maximální zanoření XML	12 úrovní

Tabulka 11: Parametry testovacího XML dokumentu

5.1 Srovnání jiných implementací

Následuje celkem 13 testů, jejichž účelem je jednak prezentace většiny implementovaných konstrukcí jazyka XQuery a jednak porovnání délek běhu s jinými dostupnými procesory.

Pro srovnání byly vybrány implementace Saxon [10] a Xml Prime [2]. Oba tyto procesory jsou volně dostupné pro nekomerční použití. Každý dotaz byl na všech implementacích spuštěn celkem 5 krát, aby nedošlo k zavádějícím výsledkům kvůli náhodným odchylkám. Časy byly měřené ve všech případech pouze na vykonávání dotazu, tzn. bez parsování vstupního XML a bez parsování, kompilace a optimalizace dotazu. Ukázalo se, že Xml Prime vyhodnocuje některé části dotazu až během iterování přes výslednou sekvenci. Do času je tedy zahrnuta jedna prázdná iterace přes výsledek.

Součástí každého testu je vstupní dotaz a dvě tabulky. První vždy udává výsledné časy v milisekundách. Řádek *Procesor* přestavuje časy pro náš implementovaný procesor. Byly měřeny časy prvního a druhého běhu dotazu. U všech procesorů je při druhém běhu evidentní využívání předalokovaných objektů z běhu prvního. Jsou uvedeny vždy nejmenší zjištěné časy.

Druhá tabulka se váže k našemu procesoru a slouží jako přehled počtu vytvořených instancí pro jednotlivé typy datového modelu v prvním běhu. Sloupec *Req* představuje počet požadavků na vytvoření objektu daného typu, sloupec *New* udává, kolik instancí bylo skutečně vytvořeno (viz. recyklace instancí v kap. 4.4.2).

Vstupní dotaz je uveden v takovém tvaru, aby vyhověl našemu procesoru. Při spouštění dotazů na procesorech podle standardu W3C je potřeba v některých případech upravit názvy funkcí – obvykle místo `distinct` použít `fn:distinct-values`.

Test 1: *Test generování XML ulzů*

```
for $i in 1 to 100000
return <a>{$i}</a>
```

	1. běh	2. běh
Procesor	324 ms	84 ms
Saxon	375 ms	297 ms
Xml Prime	906 ms	687 ms

	Req	New		Req	New
sekvence	400 002	400 002	boolean	0	0
n-tice	200 000	200 000	Xml element	100 000	100 000
tabulka	2	2	Xml atribut	0	0
integer	100 000	100 000	Xml text	100 000	100 000
double	0	0	Xml dokument	0	0
string	0	0	Xml komentář	0	0

Tento test je zaměřený především na generování většího množství DOM objektů, v tomto případě XML elementů. Během vyhodnocování musel náš procesor převést relativně velké množství celočíselných položek na n-tice a vytvořit mnoho dílčích sekvencí pro sestavení obsahu výsledných elementů.

Test 2: *Generování prvočísel*

```
for $i in 2 to 1000
where every $x in (2 to $i - 1) satisfies ($i mod $x != 0)
return $i
```

	1. běh	2. běh
Procesor	115 ms	112 ms
Saxon	63 ms	15 ms
Xml Prime	187 ms	47 ms

	Req	New		Req	New
sekvence	500 669	1 999	boolean	79 021	1
n-tice	999 000	3 994	Xml element	0	0
tabulka	2 000	4	Xml atribut	0	0
integer	57 8521	1 999	Xml text	0	0
double	0	0	Xml dokument	0	0
string	0	0	Xml komentář	0	0

V tomto případě byl testován zanořený kvantifikovaný výraz. Ve druhém běhu procesory Saxon a Xml Prime pravděpodobně lépe využily předalokované paměti.

Test 3: *Jednoduchý XPath se jmenným testem a navigací na descendant-or-self*

```
//item
```

	1. běh	2. běh
Procesor	21 ms	20 ms
Saxon	47 ms	0 ms
Xml Prime	109 ms	0 ms

	Req	New		Req	New
sekvence	1	1	boolean	0	0
n-tice	0	0	Xml element	0	0
tabulka	0	0	Xml atribut	0	0
integer	0	0	Xml text	0	0
double	0	0	Xml dokument	0	0
string	0	0	Xml komentář	0	0

Z tabulky vzniklých instancí je vidět, že bylo potřeba pouze jediné (výstupní) sekvence. Všechny XML uzly byly vytvořeny během parsování XML. Uzlů *item* je v dokumentu celkem 6 586.

Test 4: *Iterování sekvencí položek*

```
for $item in //item
return $item
```

	1. běh	2. běh
Procesor	30 ms	23 ms
Saxon	47 ms	0 ms
Xml Prime	109 ms	0 ms

	Req	New		Req	New
sekvence	6 588	6 588	boolean	0	0
n-tice	13 172	13 172	Xml element	0	0
tabulka	2	2	Xml atribut	0	0
integer	0	0	Xml text	0	0
double	0	0	Xml dokument	0	0
string	0	0	Xml komentář	0	0

Jedná se o podobný dotaz jako v předchozím případě. Zde je ovšem rozdíl v tom, že procesor musí položky ze vstupní sekvence z důvodu FLWOR výrazu převést na n-tice. Předchozí dotaz byl v procesorech Saxon a Xml Prime pravděpodobně normalizován, takže XPath musel být převeden na FLWOR. To by mohlo vysvětlovat, proč jsou výsledky u Saxonu a Xml Prime v porovnání s předchozím testem velmi podobné, zatímco u našeho procesor je pozorovatelná malá změna.

Test 5: *FLWOR v kombinaci s XPath*

```
for $item in //item
where $item/@id = "item1853"
return $item
```

	1. běh	2. běh
Procesor	31 ms	25 ms
Saxon	109 ms	0 ms
Xml Prime	156 ms	0 ms

	Req	New		Req	New
sekvence	6 589	3	boolean	6 586	1
n-tice	13 172	13 172	Xml element	0	0
tabulka	2	2	Xml atribut	0	0
integer	0	0	Xml text	0	0
double	0	0	Xml dokument	0	0
string	0	0	Xml komentář	0	0

Tento test ukazuje na rychlostní převahu našeho procesoru při prvním běhu a naopak ztrátu při druhém běhu. Saxon a Xml Prime zřejmě dokáží využít nějakých informací z předchozího běhu dotazu.

Test 6: *Dotaz s kvantifikovaným výrazem*

```
for $item in //item[location = "United States"]
where some $cat in $item/incategory/@category satisfies $cat = "category418"
return $item
```

	1. běh	2. běh
Procesor	69 ms	57 ms
Saxon	125 ms	31 ms
Xml Prime	187 ms	31 ms

	Req	New		Req	New
sekvence	21 284	4 912	boolean	29 653	1
n-tice	59 394	22 978	Xml element	0	0
tabulka	9 791	7	Xml atribut	0	0
integer	0	0	Xml text	0	0
double	0	0	Xml dokument	0	0
string	0	0	Xml komentář	0	0

Testovací dotaz 6 prezentuje možnosti kvantifikovaných výrazů. První běh byl přibližně dvakrát kratší oproti Saxonu a XmlPrime, ve druhém běhu ale byla ztráta.

Test 7: *Ukázka použití přímých konstruktorů*

```
distinct(
  for $item in //item[location = "United States"]
  let $location := $item/location/text()
  let $quantity := $item/quantity/text() cast as double
  order by $quantity
  return <item> Location: { $location }, Price: { $quantity * 24.57 } </item>
)
```

	1. běh	2. běh
Procesor	194 ms	110 ms
Saxon	359 ms	140 ms
Xml Prime	375 ms	78 ms

	Req	New		Req	New
sekvence	193 209	186 623	boolean	6 586	1
n-tice	52 316	52 316	Xml element	4 893	4 893
tabulky	7	7	Xml atribut	0	0
integer	0	0	Xml text	19 572	19 572
double	9 786	9 786	Xml dokument	0	0
string	0	0	Xml komentář	0	0

Dotaz ukazuje použití přímých konstruktorů prokládaných vnořenými výpočty. Jedná se o výpočet ceny položky na základě jednotkové ceny a množství.

Test 8: *Ukázka počítání souhrnů*

```
for $region in /site/regions
for $item in $region//item
let $count := count($item//incategory)
order by $count
return <item id = "{ $item/@id }" count = "{ $count }"/>
```

	1. běh	2. běh
Procesor	168 ms	82 ms
Saxon	218 ms	78 ms
Xml Prime	421 ms	125 ms

	Req	New		Req	New
sekvence	245 840	219 496	boolean	0	0
n-tice	39 518	39 518	Xml element	6 586	6 586
tabulka	6	6	Xml atribut	13 172	13 172
integer	6 586	6 586	Xml text	0	0
double	0	0	Xml dokument	0	0
string	0	0	Xml komentář	0	0

Tento dotaz ukazuje jednak použití dvou klauzulí `for` a jednak praktické počítání souhrnů pomocí funkce `count`.

Test 9: *Počítání souhrnů s generováním XML*

```
<summary>
{
  for $location in distinct(//location/text())
  let $count := count(//item[location = $location])
  return
  <item>
    <location>{$location}</location>
    <count>{$count}</count>
  </item>
}
</summary>
```

	1. běh	2. běh
Procesor	9 565 ms	8 094 ms
Saxon	1 687 ms	1 531 ms
Xml Prime	343 ms	79 ms

	Req	New		Req	New
sekvence	1 537 562	9 610	boolean	1 527 952	1
n-tice	3 056 832	3 056 832	Xml element	697	697
tabulka	467	467	Xml atribut	0	0
integer	232	232	Xml text	930	930
double	0	0	Xml dokument	0	0
string	0	0	Xml komentář	0	0

V tomto testu se bohužel náš procesor ukázal jako neefektivní v porovnání se Saxonem a Xml Prime. Spíše než na špatnou implementaci toto ukazuje na chybějící optimalizaci. Ke zlepšení by mělo vést zavedení operátoru GroupBy (kapitola 3.4.6). Srovnání s procesorem Xml Prime ale nemusí být úplně korektní, jelikož testováním se ukazuje, že Xml Prime provádí skutečné vyhodnocení některých zanořených částí výsledného XML stromu dotazu až při fyzickém čtení hodnot. Obsahy uzlů *location* a *count* zůstaly pravděpodobně nevypočtené.

Test 10: *Ukázka podmíněných výrazů*

```
for $item in //item
let $info :=
  if ($item/location = "United States") then
    $item/payment
  else
    $item/quantity
let $name := $item/name
return <item name="{$name}" info="{$info}"/>
```

	1. běh	2. běh
Procesor	116 ms	79 ms
Saxon	187 ms	62 ms
Xml Prime	390 ms	109 ms

	Req	New		Req	New
sekvence	59 276	39 518	boolean	6 586	1
n-tice	39 516	39 516	Xml element	6 586	6 586
tabulka	4	4	Xml atribut	13 172	13 172
integer	0	0	Xml text	0	0
double	0	0	Xml dokument	0	0
string	0	0	Xml komentář	0	0

Ukázka má spíše prezentovat možnost použití podmíněných výrazů.

Test 11: *Použití konstrukce* `typeswitch`

```
for $item in (1, "a", 2.78, "true" cast as boolean, 10)
let $typeInfo :=
  typeswitch ($item)
  case string return "string "
  case integer return "integer "
  case double return "double "
  case boolean return "boolean "
  default return "undefined "
return $typeInfo
```

	1. běh	2. běh
Procesor	0 ms	0 ms
Saxon	31 ms	0 ms
Xml Prime	140 ms	0 ms

	Req	New		Req	New
sekvence	17	17	boolean	13	2
n-tice	30	30	Xml element	0	0
tabulka	3	3	Xml atribut	0	0
integer	0	0	Xml text	0	0
double	0	0	Xml dokument	0	0
string	0	0	Xml komentář	0	0

Jedná se o velmi jednoduchý dotaz, který spíše demonstruje použití konstrukce `typeswitch`. V našem procesoru je testování datových typů podporováno pouze částečně – je možné testovat položky na typ skalární hodnoty. Relativně zajímavé je časové srovnání, které ukazuje na režijní záležitosti ohledně každého dotazu.

Test 12: *Ukázka generování přehledového XML*

```
for $location in distinct(//location/text())
return
  <region>
    <location>{$location}</location>
    <items>
      { for $item in //item[location = $location]
        return <item id="{ $item/@id }" /> }
    </items>
  </region>
```

	1. běh	2. běh
Procesor	9 905 ms	8 140 ms
Saxon	1 750 ms	1 547 ms
Xml Prime	734 ms	344 ms

	Req	New		Req	New
sekvence	1 570 952	29 828	boolean	1 527 952	1
n-tice	3 069 540	3 069 540	Xml element	7 282	7 282
tabulka	930	930	Xml atribut	6 586	6 586
integer	0	0	Xml text	1 160	1 160
double	0	0	Xml dokument	0	0
string	0	0	Xml komentář	0	0

Jedná se podobně problematický dotaz jako v testu 9. Zde je opět patrná vysoká ztráta na Saxon a Xml Prime. Dotaz ukazuje možnost generování složitějšího souhrnového XML.

Test 13: *Použití třízení ve FLWOR výrazu*

```
for $item in //item
let $name := $item/name/text()
order by $name
return $item
```

	1. běh	2. běh
Procesor	87 ms	64 ms
Saxon	203 ms	78 ms
Xml Prime	219 ms	16 ms

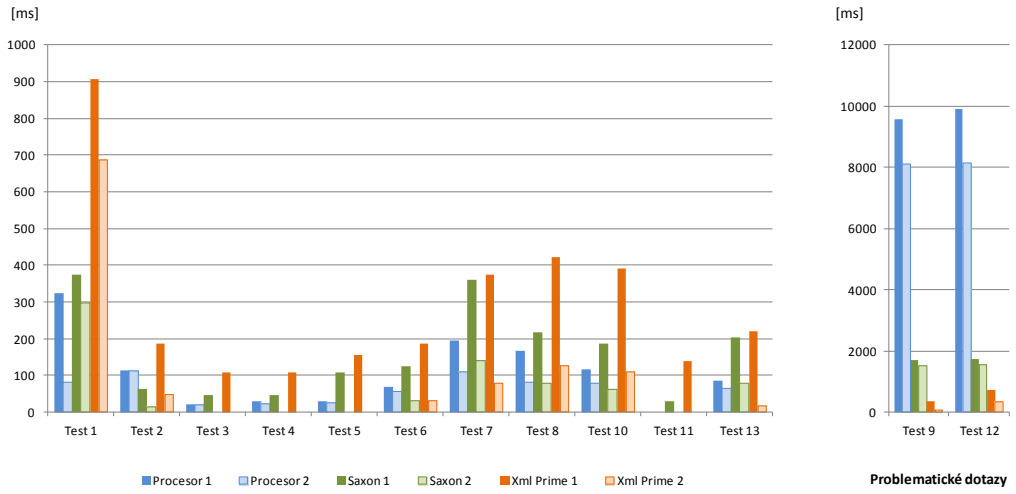
	Req	New		Req	New
sekvence	19 760	19 760	boolean	0	0
n-tice	39 516	39 516	Xml element	0	0
tabulka	4	4	Xml atribut	0	0
integer	0	0	Xml text	0	0
double	0	0	Xml dokument	0	0
string	0	0	Xml komentář	0	0

Posledním testem je ukázka třízení položek ve FLWOR výrazu. V současné implementaci se pro třízení používá algoritmus Quick sort. Pro lexikografické třízení by minimálně stály za vyzkoušení i jiné algoritmy, např. Radix sort.

5.1.1 Shrnutí časových srovnání

Testováním se ukázalo, že náš procesor ve většině případů podává srovnatelné nebo dokonce lepší výsledky než existující implementace. Grafické porovnání výsledků je znázorněno na obr. **??**. Na druhou stranu byly i dotazy (9 a 12), kde byla evidentní velká ztráta. Tato ztráta ale spíše než na možnost špatné implementace datových struktur ukazuje na chybějící optimalizaci spojující vnitřní zanořený FLWOR s vnějším FLWOR nebo XPath. U procesorů Saxon a Xml Prime je potřeba zohlednit jejich dlouholetý vývoj (Xml Prime od roku 2009, Saxon od r. 1998) a spolupráci většího množství lidí nejen na samotné implementaci, ale také na testování.

Tato implementace minimálně otevírá možnosti k dalším optimalizacím. Další možnosti zlepšení efektivity se skrývají například v používání sofistikovanějších metod pro evaluaci XPath výrazů, kdy jednotlivé části nejsou zpracovávány postupně, ale na XPath je pohlíženo jako na celek (kap. 3.4.6). Zajímavou možností by mohl být např. další překlad z plánu vykonávání přímo do strojového kódu.



Obrázek 29: Srovnání časů jednotlivých procesorů

5.2 Vliv optimalizací na délku běhu

Následující dva testy jsou zaměřeny na vliv optimalizací nadefinovaných v kapitole 3.4 na dobu vyhodnocování.

Test 14: *Optimalizace spojením kroků XPath*

Tento test ukazuje srovnání doby vykonávání bez a s optimalizací spojením kroků XPath (kap. 3.4.5). Jako ukázkový dotaz slouží dotaz z testu 8.

```
for $region in /site/regions
for $item in $region//item
let $count := count($item//incategory)
order by $count
return <item id = "{ $item/@id }" count = "{ $count }"/>
```

	1. běh	2. běh
Bez optimalizace	298 ms	192 ms
S optimalizací	168 ms	82 ms

Z výsledků je viditelný podstatný rozdíl, zejména při druhém běhu dotazu.

Test 15: *Optimalizace odstraněním* MapConcat *a* MapToItem

V tomto testu je srovnána doba vykonávání bez a s optimalizací odstraněním MapConcat a MapToItem (kap. 3.4.1 a 3.4.2). Jako ukázkový dotaz je využit dotaz z testu 2.

```
for $i in 2 to 1000
where every $x in (2 to $i - 1) satisfies ($i mod $x != 0)
return $i
```

	1. běh	2. běh
Bez optimalizace	258 ms	255 ms
S optimalizací	115 ms	112 ms

Z výsledků je pozorovatelné téměř dvojnásobné zlepšení výkonu procesoru.

6 Závěr

Výsledkem práce je funkční algebraický XQuery procesor, který vychází ze specifikace W3C, podporuje základní konstrukce jazyka XQuery od aritmetických operací přes logické operace, práci se sekvencemi pomocí FLWOR a XPath výrazů až po konstrukce jako alternativy a větvení dle datového typu.

6.1 Vlastní přínos a možnosti rozšíření

Vlastním přínosem této práce je především modulárně navržený systém, jehož části mohou být v budoucnu dále rozšiřovány větším týmem lidí. Naimplementovaný procesor přistupuje ke kompilaci XQuery dotazů bez předchozí normalizace, což může usnadnit např. vyhledávání stromových vzorů pro efektivní vykonávání na indexovaných databázích. Dále se zde objevují specifické optimalizace (kapitola 3.4) a návrhy algoritmů na řešení navigací v XML dokumentech (kapitola 4.8.3).

Tato práce otevírá velké možnosti pro vývoj a testování různých optimalizačních postupů a fyzických algoritmů řešících funkčnost jednotlivých operátorů. Reálným cílem je využívání TPQ pro vyhodnocování výrazů FLWOR podle [14]. Přínosem je také samotný naimplementovaný procesor, který je pro běžné účely prakticky použitelný.

6.2 Osobní zhodnocení

Jelikož se již delší dobu prakticky pohybuji v oblasti vývoje informačních systémů (bakalářská práce [12]) od návrhu a tvorby databází, přes implementaci GUI, až po komunikaci s koncovými zákazníky, bylo mým úmyslem vybrat si takové téma, které by mi umožnilo nahlédnout do pozadí technologií, které více nebo méně používám. Původním záměrem bylo vytvořit funkční procesor SQL, avšak vzhledem k relativní nasycenosti trhu standardními relačními databázovými stroji byla volba XQuery minimálně z hlediska studia zajímavější.

Nejsložitějším úkolem na celé práci bylo zorientovat se ve velkém množství informací ohledně XML technologií. Jak bylo v kapitole 2.3.4 uvedeno, XQuery úzce souvisí s jinými dotazovacími jazyky, takže krom studia standardu XQuery bylo potřeba nahlížet také do specifikací XPath a XML. Podrobné detailní zkoumání všech těchto standardů by vyžadovalo mnohem delší dobu a spolupráci většího množství lidí. To je důvodem, proč se tento procesor může od specifikace lišit.

Reference

[1] W3C. *W3Schools* [online]. 1999-2012 [cit. 2012-02-05]. Dostupné z: <http://www.w3schools.com>

[2] CLINICAL & BIOMEDICAL COMPUTING LTD. *XML Prime* [online]. 2009-2012 [cit. 2012-04-09]. Dostupné z: <http://www.xmlprime.com>

[3] W3C. *Extensible Markup Language (XML) 1.1 (Second Edition)* [online]. 2006 [cit. 2012-01-24]. Dostupné z: <http://www.w3.org/TR/xml11>

[4] W3C. *XML Path Language (XPath) 2.0 (Second Edition)* [online]. 2010 [cit. 2012-02-10]. Dostupné z: <http://www.w3.org/TR/xpath20>

[5] W3C. *XQuery 1.0: An XML Query Language (Second Edition)* [online]. 2010 [cit. 2012-02-03]. Dostupné z: <http://www.w3.org/TR/xquery>

[6] W3C. *XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)* [online]. 2010 [cit. 2012-03-20]. Dostupné z: <http://www.w3.org/TR/xpath-datamodel>

[7] W3C. *XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)* [online]. 2010 [cit. 2012-03-20]. Dostupné z: <http://www.w3.org/TR/xquery-semantics>

[8] WIKIPEDIE. *Lexikální analýza* [online]. 2011 [cit. 2012-01-27]. Dostupné z: <http://cs.wikipedia.org/wiki/Lexikln_analza>

[9] BŘUSKA, Filip. *Normalizace XQuery dotazů*. Ostrava, 2010. Diplomová práce. VŠB – Technická univerzita Ostrava.

[10] KAY, Michael H. *SAXON: The XSLT and XQuery Processor* [online]. 2011 [cit. 2012-02-05]. Dostupné z: <http://saxon.sourceforge.net>

[11] *A simple proof for the turing-completeness of xslt and xquery* [online]. 2006 [cit. 2012-04-11]. Dostupné z: <http://www.tcl-sfs.uni-tuebingen.de~kepser/papers/EML2004Kepser01.pdf>

[12] LUKÁŠ, Petr. *Absolvování individuální odborné praxe* [online]. Ostrava, 2010 [cit. 2012-04-11]. Bakalářská práce. VŠB – Technická univerzita Ostrava.

[13] MANOLESCU, Ioana, PAPAKONSTANTINOU a Vasilis VASSALOS. *Xml tuple algebra* [online]. 2009 [cit. 2012-04-11]. Dostupné z: <http://www-rocq.inria.fr/~manolesc/PAPERS/Encyclopedia-XMLTupleAlgebra.pdf>

[14] MICHIELS, Philippe, George A. MIHĂILĂ a Jérôm SIMÉON. *Put a tree pattern in your algebra* [online]. 2007 [cit. 2012-04-11]. Dostupné z: <http://win.ua.ac.be/~adrem/bibrem/pubs/michiels07treepattern.pdf>

[15] ROBIE, Jonathan, Don CHAMBERLIN a Daniela FLORESCU. *Quilt: An xml query language* [online]. 2000 [cit. 2012-04-11]. Dostupné z: <http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html>

[16] RÉ, Christopher, Jérôm SIMÉON a Mary FERNÁNDEZ. *A complete and efficient algebraic compiler for xquery* [online]. 2006 [cit. 2012-04-11]. Dostupné z: <http://pages.cs.wisc.edu/~chrisre/papers/ICDE06_compiler.pdf>

[17] *XMark-An XML Benchmark Project* [online]. [cit. 2012-04-25]. Dostupné z: <http://www.xml-benchmark.org/>

A Vestavěné funkce

add

Dvouparametrová funkce pro sčítání číselných položek nebo spojování textových řetězců.

add$(a, b) \rightarrow c$

a	b	c	
integer	integer	integer	součet čísel
integer	double	double	součet čísel
double	integer	double	součet čísel
double	integer	double	součet čísel
string	sequence	string	zřetězení po serializaci sekvence
string	string	string	zřetězení

sub

Dvouparametrová funkce pro odčítání číselných položek.

sub$(a, b) \rightarrow c$

a	b	c	
integer	integer	integer	rozdíl čísel
integer	double	double	rozdíl čísel
double	integer	double	rozdíl čísel
double	integer	double	rozdíl čísel

mul

Dvouparametrová funkce pro násobení číselných položek.

mul$(a, b) \rightarrow c$

a	b	c	
integer	integer	integer	součin čísel
integer	double	double	součin čísel
double	integer	double	součin čísel
double	integer	double	součin čísel

div

Dvouparametrová funkce pro desetinné dělení číselných položek.

div$(a, b) \rightarrow c$

a	*b*	*c*	
`integer`	`integer`	`double`	podíl čísel
`integer`	`double`	`double`	podíl čísel
`double`	`integer`	`double`	podíl čísel
`double`	`integer`	`double`	podíl čísel

idiv

Dvouparametrová funkce pro celočíselné dělení číselných položek.

idiv$(a, b) \rightarrow c$

a	*b*	*c*	
`integer`	`integer`	`integer`	podíl čísel bez desetinné části

mod

Dvouparametrová funkce pro zbytek po celočíselném dělení.

mod$(a, b) \rightarrow c$

a	*b*	*c*	
`integer`	`integer`	`integer`	zbytek po celočíselném dělení

veq, vneq

Dvouparametrové funkce pro hodnotové porovnávání. Funkce `vneq` vrací vždy negovaný výsledek `veq`.

veq$(a, b) \rightarrow c$, vneq$(a, b) \rightarrow c$

a	*b*	*c*	
integer	integer	boolean	porovnání celých čísel
integer	double	boolean	porovnání celého a desetinného čísla
integer	boolean	boolean	porovnání celého čísla a booleovské hodnoty převedené na 1 nebo 0
double	integer	boolean	porovnání desetinného a celého čísla
double	double	boolean	porovnání dvou desetinných čísel
string	string	boolean	porovnání celých čísel
string	xml-attribute	boolean	porovnání textového řetězce a hodnoty atributu
string	xml-element	boolean	porovnání textového řetězce s textovým obsahem elementu
string	xml-text	boolean	porovnání textového řetězce s obsahem textového uzlu
string	xml-comment	boolean	porovnání textového řetězce s komentářem
xml-attribute	xml-attribute	boolean	porovnání hodnot dvou atributů
xml-attribute	string	boolean	porovnání hodnoty atributu a textového řetězce
xml-attribute	xml-element	boolean	porovnání hodnoty atributu s textovým obsahem elementu
xml-attribute	xml-text	boolean	porovnání hodnoty atributu s obsahem textového uzlu
xml-attribute	xml-comment	boolean	porovnání hodnoty atributu s komentářem
xml-element	xml-element	boolean	porovnání struktury celých elemtnů včetně obsahu
xml-element	xml-attribute	boolean	porovnání textového obsahu elementu s hodnotou atributu
xml-element	string	boolean	porovnání textového obsahu elementu s textovým řetězcem
xml-element	xml-text	boolean	porovnání textového obsahu elementu s obsahem textového uzlu
xml-element	xml-comment	boolean	porovnání textového obsahu elementu s komentářem
xml-text	xml-text	boolean	porovnání textů dvou textových uzlů
xml-text	xml-element	boolean	porovnání obsahu textového uzlu s textovým obsahem elementu
xml-text	xml-attribute	boolean	porovnání obsahu textového uzlu s hodnotou atributu
xml-text	string	boolean	porovnání obsahu textového uzlu s textovým řetězcem
xml-text	xml-comment	boolean	porovnání textového obsahu textového uzlu s komentářem
xml-comment	xml-comment	boolean	porovnání dvou komentářů
xml-comment	xml-text	boolean	porovnání komentáře a obsahu textového uzlu
xml-comment	xml-element	boolean	porovnání komentáře s textovým obsahem elementu
xml-comment	xml-attribute	boolean	porovnání komentáře s hodnotou atributu
xml-comment	string	boolean	porovnání komentáře s textovým řetězcem

vlt, vgt, vle, vge

Dvouparametrové funkce pro hodnotové uspořádání datových položek. Funkce `vgt` odpovídá $<$, `vlt` odpovídá $>$, `vge` odpovídá $\geq$, `vle` odpovídá $\leq$.

$$\text{vlt}(a, b) \rightarrow c, \text{vgt}(a, b) \rightarrow c, \text{vle}(a, b) \rightarrow c, \text{vge}(a, b) \rightarrow c$$

a	*b*	*c*	
integer	integer	boolean	uspořádání celých čísel
integer	double	boolean	uspořádání celého a desetinného čísla
integer	boolean	boolean	uspořádání celého čísla a booleovské hodnoty převedené na 1 nebo 0
double	integer	boolean	uspořádání desetinného a celého čísla
double	double	boolean	uspořádání dvou desetinných čísel
string	string	boolean	uspořádání celých čísel
string	xml-attribute	boolean	lexikografické uspořádání textového řetězce a hodnoty atributu
string	xml-element	boolean	lexikografické uspořádání textového řetězce s textovým obsahem elementu
string	xml-text	boolean	lexikografické uspořádání textového řetězce s obsahem textového uzlu
string	xml-comment	boolean	lexikografické uspořádání textového řetězce s komentářem
xml-attribute	xml-attribute	boolean	lexikografické uspořádání hodnot dvou atributů
xml-attribute	string	boolean	lexikografické uspořádání hodnoty atributu a textového řetězce
xml-attribute	xml-element	boolean	lexikografické uspořádání hodnoty atributu s textovým obsahem elementu
xml-attribute	xml-text	boolean	lexikografické uspořádání hodnoty atributu s obsahem textového uzlu
xml-attribute	xml-comment	boolean	lexikografické uspořádání hodnoty atributu s komentářem
xml-element	xml-element	boolean	lexikografické uspořádání textového obsahu dvou elementů
xml-element	xml-attribute	boolean	lexikografické uspořádání textového obsahu elementu s hodnotou atributu
xml-element	string	boolean	lexikografické uspořádání textového obsahu elementu s textovým řetězcem
xml-element	xml-text	boolean	lexikografické uspořádání textového obsahu elementu s obsahem textového uzlu
xml-element	xml-comment	boolean	lexikografické uspořádání textového obsahu elementu s komentářem
xml-text	xml-text	boolean	lexikografické uspořádání textů dvou textových uzlů
xml-text	xml-element	boolean	lexikografické uspořádání obsahu textového uzlu s textovým obsahem elementu
xml-text	xml-attribute	boolean	lexikografické uspořádání obsahu textového uzlu s hodnotou atributu
xml-text	string	boolean	lexikografické uspořádání obsahu textového uzlu s textovým řetězcem
xml-text	xml-comment	boolean	lexikografické uspořádání textového obsahu textového uzlu s komentářem
xml-comment	xml-comment	boolean	lexikografické uspořádání dvou komentářů
xml-comment	xml-text	boolean	lexikografické uspořádání komentáře a obsahu textového uzlu
xml-comment	xml-element	boolean	lexikografické uspořádání komentáře s textovým obsahem elementu
xml-comment	xml-attribute	boolean	lexikografické uspořádání komentáře s hodnotou atributu
xml-comment	string	boolean	lexikografické uspořádání komentáře s textovým řetězcem

eq, eq

Dvouparametrové funkce pro obecné porovnávání. Pro jednoprvkové sekvence pracují funkce stejně jako `veq` a `vneq`, pro víceprvkové sekvence vrací funkce pravdivou hodnotu, pokud porovnání vyhoví alespoň jedné dvojici položek z obou sekvencí.

$\text{eq}(a, b) \to c, \text{neq}(a, b) \to c$

a	*b*	*c*	
sequence	sequence	boolean	obecné porovnání sekvencí

lt, gt, le, ge

Dvouparametrové funkce pro obecné porovnávání. Pro jednoprvkové sekvence pracují funkce stejně jako `vlt`, `vgt`, `vle` a `vge`, pro víceprvkové sekvence vrací funkce pravdivou hodnotu, pokud porovnání vyhoví alespoň jedné dvojici položek z obou sekvencí.

lt$(a, b) \rightarrow c$, gt$(a, b) \rightarrow c$, le$(a, b) \rightarrow c$, ge$(a, b) \rightarrow c$

a	*b*	*c*	
`sequence`	`sequence`	`boolean`	obecné porovnání sekvencí

abs

Absolutní hodnota číselné položky.

abs$(a) \rightarrow b$

a	*b*	
`integer`	`integer`	absolutní hodnota celého čísla
`double`	`double`	absolutní hodnota desetinného čísla

count

Počet položek v sekvenci.

count$(a) \rightarrow b$

a	*b*	
`sequence`	`integer`	počet položek v sekvenci

position

Pozice kontextového uzlu v rámci sekvence položek, ke které náleží (číslováno od 1).

position() $\rightarrow a$

a	
`integer`	pozice kontextového uzlu

position

Délka sekvence, které náleží kontextový uzel.

last() $\rightarrow a$

a	
`integer`	pozice kontextového uzlu

doc

Načtení externího XML dokumentu. Pro čtení XML a převod na DOM je využit Xerces Parser. Funkce argumentem přebírá textový řetězec reprezentující cestu k souboru.

doc($fileName$) → *a*

fileName	*a*	
`string`	`xml-document`	načtený XML dokument

root

Funkce vrací vstupní XML dokument, který je načten a zpracován ještě před začátkem vyhodnocování dotazu.

root() → *a*

a	
`xml-document`	vstupní XML dokument

predicate

Funkce slouží pro vyhodnocení predikátů XPath výrazů (viz. 3.3.3).

predicate(*a*) → *b*

a	*b*	
`sequence`	`boolean`	efektivní booleovská hodnota
`integer`	`boolean`	booleovská hodnota informující, zda pozice kontextového uzlu odpovídá číslu *a*

boolean

Funkce vrací tzv. efektivní booleovskou hodnotu (viz. 3.3.3).

boolean(*a*) → *b*

a	*b*	
`sequence`	`boolean`	efektivní booleovská hodnota

range

Funkce utvoří sekvenci čísel od *a* do *b* s krokem 1.

range(a, b) → c

a	b	c	
`integer`	`integer`	`sequence`	sekvence čísel od a do b

distinct

Funkce na základě porovnání (funkce `eq`) provádí odstranění duplicit ze sekvence. Došlo k rozšíření všech tříd datového modelu o metodu vracející hash klíč, což umožňuje relativně efektivní činnost eliminace duplicitních hodnot podobně jako v operátoru TreeJoin.

distinct(a) → b

a	b	
`sequence`	`sequence`	výstupní sekvence bez duplicitních hodnot

avg

Funkce vrací průměrnou hodnotu ze sekvence, jejíž položky musí být explicitně přetypovatelné na desetinné číslo.

avg(a) → b

a	b	
`sequence`	`double`	průměrná hodnota

sum

Funkce vrací součet hodnot ze sekvence, jejíž položky musí být explicitně přetypovatelné na desetinné číslo.

sum(a) → b

a	b	
`sequence`	`double`	Součet

min

Funkce vrací minimální hodnotu ze sekvence. Podmínkou je, aby byly prvky v sekvenci navzájem porovnatelné.

min(a) → b

a	b	
`sequence`	`item`	minimální hodnota

min

Funkce vrací maximální hodnotu ze sekvence. Podmínkou je, aby byly prvky v sekvenci navzájem porovnatelné.

$\max(a) \rightarrow b$

a	b	
`sequence`	`item`	maximální hodnota

empty

Funkce vrací pravdivou booleovskou hodnotu, jestliže je sekvence předaná argumentem prázdná.

$\text{empty}(a) \rightarrow b$

a	b	
`sequence`	`boolean`	prázdnost sekvence

not

Jednoduchá funkce pro zjištění negované booleovské hodnoty.

$\text{not}(a) \rightarrow b$

a	b	
`boolean`	`boolean`	negovaná hodnota

B Kompatibilita datových typů

I – implicitní a explicitní přetypování t_1 na t_2

E – explicitní přetypování t_1 na t_2

$\downarrow t_1, t_2 \rightarrow$	sequence	boolean	integer	string	double	xml-attribute	xml-comment	xml-node	xml-text	xml-document
sequence	I	I[1]	I[1]	I[1]	I[1]	I[1]	I[1]	I[1]	I[1]	
boolean	I	I	E	E	E					
integer	I	E	I	E	E					
string	I	E[2]	E[3]	I	E[4]					
double	I	E	E	E	I					
xml-attribute	I	E[2]	E[3]	I	E[4]	I				
xml-comment	I	E[2]	E[3]	I	E[4]		I			
xml-node	I	E[2]	E[3]	I	E[4]			I		
xml-text	I	E[2]	E[3]	I	E[4]				I	
xml-document	I							I		I

Tabulka 12: Kompatibilita datových typů

[1] – Za předpokladu, že sekvence obsahuje jedinou položku.

[2] – Za předpokladu, že textový řetězec, hodnota atributu, komentáře, textového uzlu nebo textového obsahu elementu je “0”, “1”, “true” nebo “false”.

[3] – Za předpokladu, že textový řetězec, hodnota atributu, komentáře, textového uzlu nebo textového obsahu elementu reprezentuje celé číslo.

[4] – Za předpokladu, že textový řetězec, hodnota atributu, komentáře, textového uzlu nebo textového obsahu elementu reprezentuje desetinné číslo.

C Spustitelný procesor

Samostatně spustitelný procesor je možné nalézt na přiloženém CD k této práci. Jeho obsluha je snadná:

```
XQueryProcessor.exe <soubor XML> <souboru s dotazem> [<výstupní soubor>]
```

Specifikace výstupního souboru je nepovinná, v případě, že výstup není definován, vypíše se výsledek dotazu přímo na monitor.

Spustitelný soubor pouze demonstruje jedno z využití procesoru. Součástí přiloženého CD jsou zdrojové soubory s projektem pro Microsoft Visual Studio 2005. Praktické uplatnění spočívá ve využití techto zdrojových souborů jako komponenty pro implementaci jiných rozsáhlejších projektů využívajících možnosti vyhledávání v XML. Obsah CD je blíže specifikován souborem „Obsah.pdf“, který je umístěn v kořenovém adresáři.